DISK SYSTEMS FOR THE BBC MICRO

IAN SINCLAIR

Disk Systems for the BBC Micro

Disk Systems for the BBC Micro

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984
Reprinted 1984 (twice)

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Disk systems for the BBC Micro.
1. BBC Microcomputer 2. Floppy disks
(Computer storage devices)
I. Title
001.64'42 QA76.8.B3

ISBN 0-246-12325-7

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

At some time or another, a serious programmer will find that the use of cassettes is intolerable, and will turn to disk drives. Since the BBC 'B' machine is bought mainly by serious programmers, it is inevitable that a very large proportion of BBC 'B' users have acquired, are acquiring, or intend to acquire, disk systems. The official disk system from Acorn Computers, however, has not always been readily available, and several competing upgrades now exist.

This book is intended as a beginner's introduction to the disk systems that are available for the BBC 'B' machine. By 'beginner', I don't necessarily mean a beginner to the use of the BBC Micro, but a beginner to disk systems. At the same time, I hope that this book will be intelligible and useful to the beginner who has launched into computing with a disk-equipped BBC Micro. The main feature of this book, then, will be greatly extended explanations of what disk operation is about, and how to make the most effective use of disks. This is not always apparent to the newcomer to disk systems, even after considerable experience of the use of the cassette-based machine. I shall not assume, as so many books on disk operation seem to assume, that the reader is at ease with machine code or hexadecimal notation, so these points will be explained as they are introduced. The book is not, however, intended as a do-it-yourself guide to hardware, so no details of how to install a disk system are included.

The use of 'systems' in the plural is intentional. At the time of writing, users could buy the official Acorn disk system or choose between two others, the PACE/AMCOM and the Watford systems. These alternatives were introduced in order to satisfy a large demand for disk systems which was not being met fast enough, and they have added some commands which are not available on the Acorn system. I should point out, however, that this book does not attempt in any way to sell one system at the expense of another. What I want to do is to point out the differences between the systems, so that the reader can make an informed choice. At the time of writing, more drastic add-on systems, which entailed the use of additional circuit boards, were being announced. These are intended to replace the rather out-of-date and scarce 8271 chip which the BBC Micro uses to control its disk system. This replacement has the added advantage of permitting 'double-density operation', meaning that much more data can be packed on to each disk. Upgrades of this type are *not* covered in this book, mainly because samples were not available at the time of writing, but also because they involve much greater changes to the machine itself.

The book also covers the use of some 'disk utilities' – programs (on disk) which can provide editing actions for disks. The beginner generally finds the action of these utilities very confusing. I have therefore included some examples of how such utilities can be of very great help – such as in reading a disk that was made by a different machine.

As always, this book owes its creation to a number of people. I would particularly like to thank Kevin Gibson, of PACE Software, for many interesting discussions on the PACE/AMCOM system. I would also like to thank Nasir Jessa, of Watford Electronics, for the Watford DFS chip, and the manual describing its use. I am also indebted to the Quentin Bell Organisation for a copy of the Acorn DFS manual. Without their generous cooperation, the book would undoubtedly be much slimmer and less useful. I must also thank my long-suffering and patient friends at Granada Publishing. Richard Miles commissioned the manuscript, David Fulton went so far as to buy himself a BBC disk machine, Sue Moore did miracles on my typescript and the typesetters and printers worked at breakneck speed to bring the book quickly into existence. I am deeply grateful to all of them. I would also like to thank Pinner Wordpro for their unfailingly rapid deliveries of disks to me each time I telephoned them. In a business where delivery dates can be decidedly flexible, their cheerful next-day service is quite outstanding.

The spelling 'disk' has been used throughout the text of this book. This is the American spelling, which has become universal. Acorn, however, use 'disc' as an option, and several suppliers of disk equipment for the BBC Micro use this spelling also. The spelling 'disc' has been used when it occurs in a menu or a catalogue display as, for example, in Fig. 6.1.

Ian Sinclair

Chapter One
About Disks and Disk Systems

Why use disks?

One of the questions that a beginner to computing inevitably asks is 'Why use disks?' The obvious reasons are not necessarily the most important ones. The novice owner will see more clearly the advantages of using disks only after he has spent some time using cassettes. We'll start, then, by showing why the use of disks is so important for the more advanced programmer and user alike. To start with, a disk offers *much faster operation*. If you use a machine to load one program, and then use that program (a game perhaps) for several hours, this speed advantage may be of little use. It certainly would not justify the cost of a disk system. On the other hand, if you are developing programs for yourself, you may want to load a program, make changes, and save it again before you try the new version out. This can be very tedious if you have to wait for cassettes to load and save. It's even more tedious because cassette operation is not automatic. You either have to store each version of the program on a new cassette, or use a long cassette (C60 or C90), with each program version noted as a starting point on the tape counter. If you use separate cassettes, you may find yourself holding a dozen of them by the time the program is complete. If you use C90s, you will need paper to note the tape count positions of each version of the program. Either way, it's tedious. Another class of user who will benefit greatly from the use of disks is the text writer. If you use the BBC Micro, as I do, to create and edit text, with WORDWISE or VIEW text editor programs, then the time that is needed for cassettes to load or save the data is a definite handicap. If you want to load a piece of text, change a few words, and then store the new version back, the loading and saving time is a very large part of the total.

The overwhelming advantage of using a disk system, however, is *automatic operation.* The BBC cassette system does, at least, permit the motor of the cassette recorder to be controlled, and it allows programs or data files to be referred to by name. If you try to load a program called "TEXTINDEX", however, without winding the cassette back to the beginning, you may find that the program cannot be loaded. This is because recording on tape is 'serial' – you start recording at the beginning of the tape, and wind it on to the end. If you then want to load something which is at the start of the tape, you have to rewind it for yourself. The computer does not control the actions of fast forward and reverse, because the cassette recorder was never intended as a way of storing computer programs and data. The disk system, by contrast, is completely computer-controlled. The only manual action is that of putting in the correct disk, and making sure that it is the right way round. On loading, the computer will then use its disk operating system to find the title of the program or other material that you want. Having located the start, it will then load the data into the computer in a time of a few seconds. Saving is just as automatic. The SAVE command is followed by a filename (and other information in some cases), and pressing RETURN carries out the actions of finding unused space on the disk, and saving the data. The automatic nature of this action also means that a 'catalogue' can be kept on the disk itself. This means that you can insert a disk and obtain information on what is stored on it without the need to play back the whole disk. Though the BBC cassette system permits cataloguing, you have to replay a whole cassette to see its catalogue.

In addition to these compelling reasons for using disks, we must add the *extra commands* that the disk operating system permits. Some computers go much further in this respect, so that their disk system adds a BASIC of its own. In the BBC disk system, the new commands are all closely tied to the use of the disk system itself, and we shall examine them in detail. Operations such as merging programs, however, are much more easily carried out when the disk system is being used than is possible with the use of cassettes. Several of the extra commands, however, allow you to obtain a lot more information about how the data is stored on the disk. This will not be of immediate use to you if you haven't used disks before, but its usefulness will be apparent before long.

Finally, the use of disks can bring *order and reliability* to what can be a very haphazard business. When you use cassettes for filing programs and data, you inevitably end up with a very large number of cassettes, all of which have to be catalogued. I had over two hundred cassettes at one stage! It can take a considerable time to locate a program on a cassette. Although a disk cannot hold quite as much information as a C90 cassette, the information is much easier to get at. This encourages you to use the whole of a disk, whereas you might use only the first ten minutes of a C90 cassette. It's quite possible to find, for example, that you can keep all of the programs that you want to use on one single disk! This is such a liberation that it almost justifies the use of disks by itself. Disks are slim and compact to store, so that a box of ten disks, holding a huge number of programs, will take up little more space than a couple of cassettes. The reliability of disk recording means that you can make a back-up copy of a valuable program, and be fairly certain that you will never need it. Unless you spill coffee all over a disk, demagnetise it or tear it apart, it's unlikely that you will lose a program. Cassettes are *never* so reliable.

What is a disk system?

'Disk system' is the name that is given to a complicated combination of hardware and software. 'Hardware' means the equipment in boxes; 'software' means the programming which can be on disk or in the form of chips that plug into the machine. A disk system comprises the disk drive (or drives), the disk controlling circuits, and the disk operating system. The disk drive is the box which contains the mechanical parts which spin the disk and make contact with its surface. These drives are available from a large number of manufacturers such as Olivetti (used for the official BBC disk system), Teac, Canon, Tandon, Mitsubishi and others. Any standard five and a quarter inch drive can be used with the BBC machine. At present, eight inch disk drives cannot be used with the standard operating system. The new three inch drives come with special provisions for fitting to the BBC disk system. Though any standard drive can be used, you can't simply take a drive from another machine and plug it in place. The drive is linked to the BBC machine by means of a data cable, which must be fitted with the correct kind of plug to engage in the disk socket at the front underside of the BBC machine. This plug must also be correctly wired. Do not, then, buy a disk drive unless it is fitted with a suitable lead and plug, or you may find yourself with a job of plug-fitting that is rather more complicated than fitting a plug to a mains lead! In general, it's getting accurate information on where each lead should go that is frustrating rather than the actual fitting of the plug.

The disk drive may come with a mains cable, or with a power cable that plugs into the BBC machine (Fig.1.1). The BBC machine has

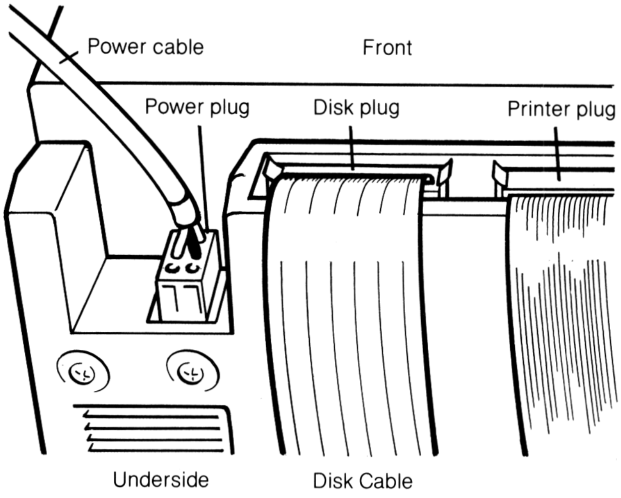


Fig. 1.1. How the cables for the disk drive(s) are fastened to the BBC machine. A 'self-powered' disk will not require the power cable connected to the BBC machine, but will need a separate power lead fitted with a 13A plug. The disk drive cable is a flat multi-way cable.

been designed with a power supply that provides extra current at low voltage for a disk drive. The early BBC machines did not have this power take-off point, which is on the underside of the case, almost under the loudspeaker aperture. It is often stated that an early power supply can be exchanged, free of charge, for the later type. I found it easier to get a new machine than to change the power supply! If you buy a disk drive which is 'self-powered', it comes with a normal mains cable, which can be fitted with the normal three-pin mains plug. There is no clear advantage or disadvantage either way. It's said that a self-powered disk drive can sometimes suffer from electrical interference carried through the mains. I think that if the disk drive and the BBC computer are plugged into the same socket strip, this problem is negligible. On the other hand, the additional current taken by the disk drive will cause the power supply of the BBC machine to run hotter when the drive is not self-powered.

The controlling circuits for the disk system are in the form of circuits on the main board of the BBC machine. When the machine is manufactured, the circuits are put in place, but the essential 'chips' that make the circuits work are not. These circuits have to send the correct signals to the disk drive unit to carry out the actions of selecting the correct point on the disk, and then reading or writing data. Ten of these chips are needed, and each one is a small block with a number (14, 16 or more) of pin connectors at each side. One of them is much larger – it is the disk controller chip, and it is labelled 8271. If you have had any experience at all of installing integrated circuit chips, you can install the disk operating chips for yourself, following closely the instructions that come with the kit of chips. You must be careful about two points. One is that each chip has a 'top' end, which is marked by an indentation. All of the chips in the BBC machine go the same way round. The other point is that all the pins on a chip must be straight if the chip is to fit in its holder. When you push a chip into place, the main board will bend, and this can cause damage. I prefer to take the main board out when fitting a large number of chips, but this is definitely not a job for anyone who has no experience of working on electronic circuits. On the whole, it's better to have the chips fitted professionally if you have any doubts at all of your ability to cope. The kits, however, generally contain good clear instructions for anyone who has had some experience.

Finally, we have the Disk Filing System (DFS). A 'file' in this sense means any collection of data which can be stored on the disk. The DFS consists of a program, and most computers use a 'DOS-disk' to hold this program. DOS is short for 'Disk Operating System'. When this is done, a lot of the RAM memory (the memory that is free for you to use) is needed for holding the DFS. The BBC machine, however, uses another chip, the DFS chip, to hold this information, and this leaves much more of the memory free. Some memory *is* used, and we'll come back to that point later. When it comes to fitting the DFS chip, you have – at the time of writing – a choice of three. One is the official Acorn chip. The others are the PACE/AMCOM chip, obtainable from PACE Software of Bradford, and the other is the Watford DFS, obtainable from Watford Electronics. Both of the alternative DFS chips can behave in the same way as the Acorn chip, but they both offer other (and differing) ways of operating which may be of advantage to some users. Full descriptions of these advantages will be covered in later chapters.

Tracks, sectors and density

The language of disk recording is very different from that of cassette recording. If your sole concern is to save and load programs in BASIC, you may possibly never need to know much about these terms. A working knowledge of how disk storage operates, however, is useful. To start with, it can clear up the problem of which disks are suitable for your drives. At a more advanced level, it can allow you to extract information from damaged disks, and to make changes to the information that is stored on disks.

Unlike tape, which is pulled past a recording/replay head, a disk spins around its centre. When you insert a disk into a drive, it is located in place, and when the drive operates, a hub engages the central hole of the disk, clamps it, and starts to spin it at a speed of about 300 revolutions per minute. The disk is a circular flat piece of plastic which has been coated with magnetic material. It is enclosed in a cardboard (or plastic) jacket to reduce the chances of damage to the surface. The hub part of the disk should also be reinforced to avoid damage when it is gripped by the drive. The surface of each disk is smooth and flat, and any physical damage, such as a fingerprint or a scratch, can cause loss of recorded data. The jacket has slots and holes cut into it so that the disk drive can touch the disk at the correct places.

Through a slot that is cut in the jacket (Fig. 1.2), the head of the disk drive can touch the surface of the disk. This head is a tiny electromagnet, and it is used both for writing data and reading. When the head writes data, electrical signals through the coils of wire in the head cause changes of magnetism. These in turn magnetise the disk surface. When the head is used for reading, the changing magnetism of the disk as it turns causes electrical signals to be generated in the coils of wire. This recording and replaying action is very similar to that of the cassette recorder, with one important difference. The cassette recorder was never designed to record digital signals from computers, but the disk head is. The reliability of recording on a disk is therefore very much better than you can ever hope for from a cassette.

Unlike the head of a cassette recorder, which does not move once it is in contact with the tape, the head of a disk drive moves quite a lot. If the head is held steady, the spinning disk will allow a circular strip of the magnetic material to be affected by the head. By moving the head in and out, to and from the centre of the disk, the drive can make contact with different circular strips of the disk. These strips

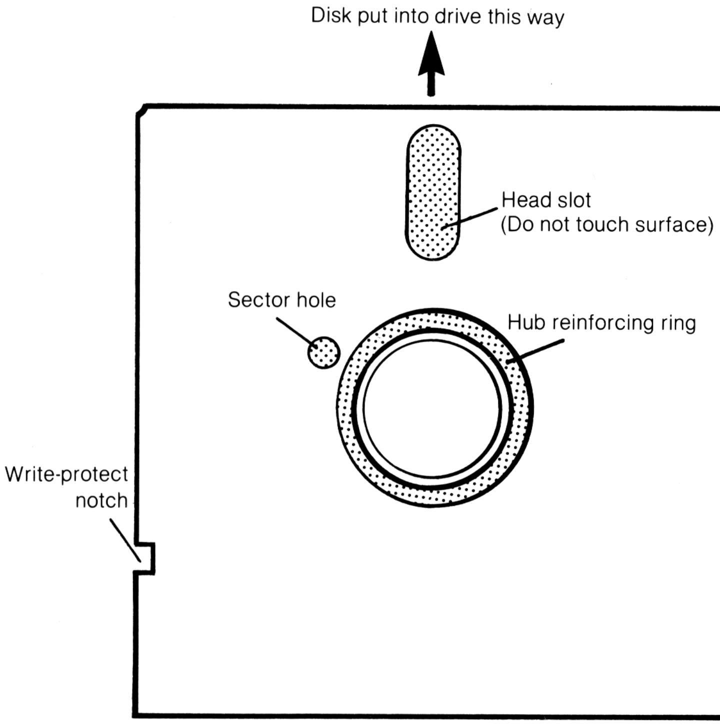


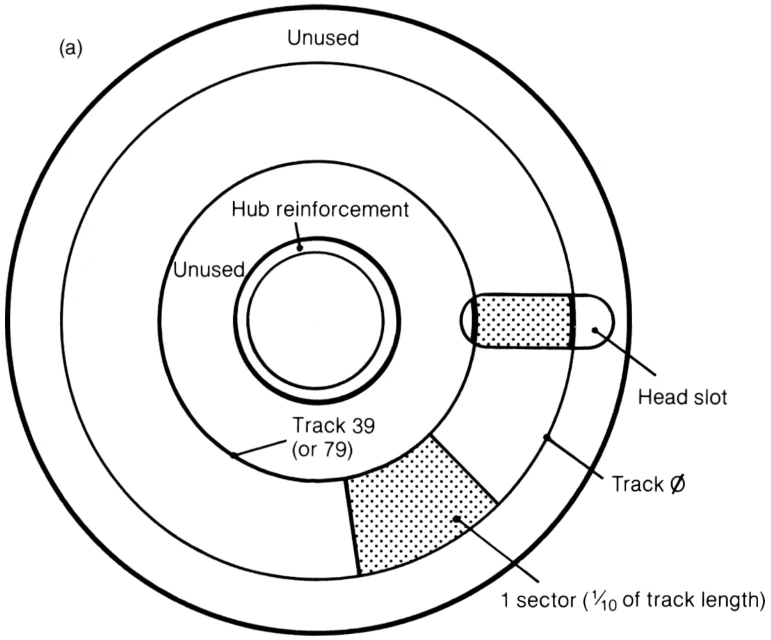
Fig. 1.2. A disk in its protective jacket. The head slot allows the read/write head of the disk drive to touch the magnetic surface and move radially across it as the disk spins. The sector hole is used to detect the start of Sector 3.

are called 'tracks'. Unlike the groove of a conventional record, these are circular, not spiral, and they are not grooves cut into the disk. The track is invisible, just as the recording on a tape is invisible. What creates the tracks is the movement of the recording/replay head of the disk drive. A rather similar situation is the choice of twin-track or four-track on cassette tapes. The same tape can be recorded with two or four tracks, depending on the heads that are used by the cassette recorder. There is nothing on the tape which guides the heads, or which indicates to you how many tracks exist.

The number of tracks, therefore, depends on your disk drives. The vast majority of drives (certainly all of the BBC drives) use either 40 or 80 tracks. Since the movement of the head is the same, the distance between tracks of the 80-track drive is smaller. Forty-track disks use 48 tracks per inch, and 80-track drives use 96 tracks per inch. This means that the positioning of the head on the disk must be more precise for an 80-track drive. In theory, this should make 80-track drives slightly less reliable, but the difference is virtually unmeasurable. The only difference is that 80-track drives should be more easily upset by dirt, smoke or other forms of contamination. At the time of writing, however, it is more common to find programs being made available on 40-track disks. Disk drives which can use either 40 or 80 tracks are available, and both the PACE/AMCOM and Watford systems allow you to read 40-track disks even on an 80-track drive. This is an important point if you have to use disks made by other users on a variety of different drives.

Once you have accepted the idea of invisible tracks, it's not quite so difficult to accept also that each track can be invisibly divided up. The reason for this is organisation – the data is divided into 'blocks', each of 256 bytes. A byte is the unit of computer data; it's the amount of memory that is needed for storing one character. Each track of the disk is divided into ten 'sectors', and each of these sectors can store 256 bytes. These figures are the same for both 40- and 80- track disks. The use of 256 bytes stored in each sector is well below the storage ability of the disk, however. Many computers allow 'double density' recording, which means that each sector has twice as much information (512 bytes) placed on it. This cannot be done with the BBC system, no matter which of the DFS chips is used, because the 8271 chip which controls the disk operation simply does not permit operation at double density. At the time of writing, at least two firms were offering add-on disk-operating boards which featured a different controller. These would support double-density operation, so doubling the amount of information that could be placed on the disk. It's too early to comment on these systems, but my experience of a similar add-on for the TRS-80 suggests that they should be reliable. The only problem is that you could end up creating disks which none of your friends could use.

The next thing that we have to consider is how the sectors are marked out. Once again, this is not a visible marking, but a magnetic one. The system is called 'soft-sectoring'. Each disk has a small hole punched into it at a distance of about 25 mm (1″) from the centre. There is a hole cut also through the disk jacket, so that when the disk is turned round, it is possible to see right through the hole when it comes round. When the disk is held in the disk drive and spun, this position can be detected using a beam of light. This is the 'marker', and the head can use this as a starting point, putting a signal on to the disk at this position, and at nine others, equally spaced, so as to form ten sectors (Fig.1.3). This sector marking has to be carried out on each track of the disk, which is part of the operation that is called 'formatting'.



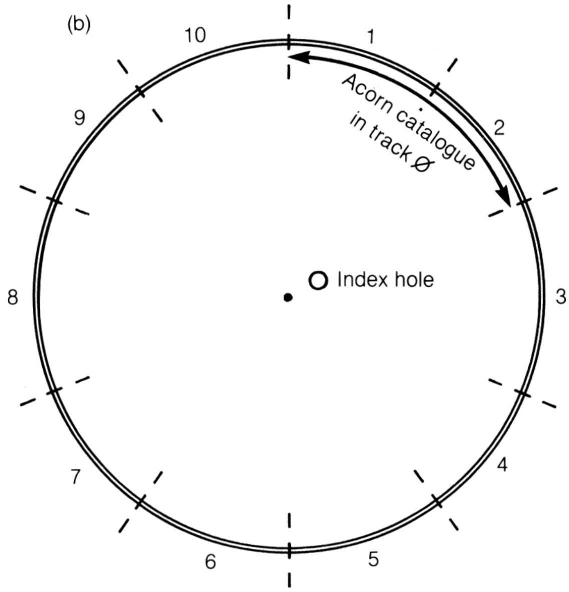


Fig. 1.3. Tracks and sectors. Only part of the disk surface is used to 'mark out' the tracks, and the same distance (about 0.83 inch) is used whether 40- or 80-tracks are used. There are always 10 sectors per track on the 5¼ inch disks.

Formatting disks

Formatting disks, as we have seen, consists partly of the action of 'marking out' the sectors on a disk. The formatting action should also, however, test the disk. This is done by writing a pattern to each sector, and checking that an identical pattern is read back later. Failure to do so indicates a faulty sector, and a disk with such a fault should be thrown away, once you are sure that the fault is genuine. That last remark needs some explanation. When disks are manufactured, they are tested. The best of the bunch can be used for the most demanding recording systems, using double density, both sides of the disk and 80 tracks. If only one side of the disk is good enough, the disk can be sold as single-sided. If the disk will not reliably record tightly packed data, it may be good enough to work with single density and with 40 tracks. Never be tempted to try to operate a disk beyond its stated limits, because it has already been tested beyond these limits and has failed! You may sometimes read descriptions (mostly copied from an article in an American magazine in 1978) of how to cut extra holes in a disk jacket and use the other side of the disk. This can work, but it is fraught with problems. One is, as we've seen, that the disk has already been judged as not being up to it. The other is that when you turn the disk round and use its other side, you are revolving the disk the other way, and this inevitably releases some dust that was quite comfortably trapped when the disk was revolving in its original direction. My advice is – don't do it!

Formatting, then, consists of marking out sectors and testing them. As each sector is checked, its number is printed on the screen, and the letter G is placed against the number if the sector is good. 'When' would be a better word than 'if', because it's very rare to find a fault at this stage. If you find a disk fault at the formatting stage, then return the disk. The way that formatting is controlled is done very differently by the different operating systems. The Acorn system uses a 'system disk'. This contains a formatting program, and you have to load this program into memory then remove the system disk. You then place a new disk into the drive, and carry out the formatting. You can specify 40 or 80 tracks, and you must be careful to specify no more than the number of tracks that your drive can use. If you have a 40-track drive and you try to format 80 tracks, there is a danger that the recording head will hit the hub or the edge of the slot in the jacket, causing damage. Use the BREAK key to stop formatting if you make a mistake like this. If you have two drives, the formatting disk can stay in one drive, or it can be removed so that you can format disks in the two drives alternately. The fact that the formatting program has to be in the memory of the computer means that any program that is in the memory will be lost (unless it has been shifted to another part of the memory). It's important, therefore, always to have a store of formatted disks in hand. If you run out of disks during programming, it's very tempting to format another disk, but this can result in losing your program! The AMCOM and the Watford disk systems have the *FORMAT command as part of the DFS chip, not loaded from another disk. It is perfectly safe, then, to format a disk while a program is contained in the computer. All of the systems, however, safeguard the disk. When a disk is formatted, any data that was stored on it is lost due to the testing process. To start with, you cannot use the *FORMAT command unless you have just previously used the *ENABLE command. This is one safeguard. In addition, if the formatting program detects data on a disk, you are reminded by a screen message, and asked if you want the disk erased. Reformatting an old disk is a useful and certain way of clearing the whole disk and testing it.

Finally, the formatting action writes on to some sectors on the first track. This portion is reserved as a way of storing information about the contents of the disk. To put it crudely, the disk system reads the first few sectors to find if a program is stored on the disk, and then to find at which sector the program starts. With this information, the head can then be moved to the start of the program, and loading can start. This part of the track is known as the catalogue. When you wipe a program or some data from the disk, all you do is remove its catalogue entry – the data remains stored on the disk until it is replaced by new data. This can sometimes allow you to recover a program that you thought you had erased. The principle is the same as the OLD command of the BBC machine itself.

Single and double-sided drives

Disk drives can be manufactured with two heads, placed almost opposite each other, so that both sides of a disk can be used. Such a double-sided drive must use double-sided disks, which have slots cut into both sides of the jacket. It's not advisable to use double-sided disks if you have only a single-sided drive, because a single-sided drive will not offer the same support to the opposite side. When double-sided drives are used, each drive side is treated by the computer's DFS as another drive, just as if it were physically separate. The computer can control up to four drives, so that twin double-sided drives are the most you can have. Figure 1.4 shows how the drives are referred to by number.

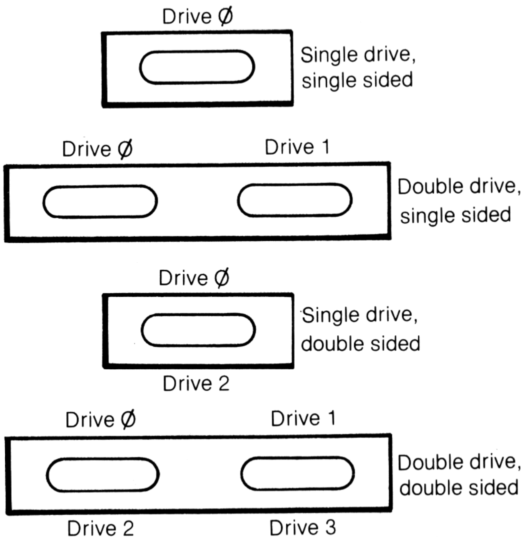

Drive Ø
Single drive, single sided
Drive Ø
Drive 1
Double drive, single sided
Drive Ø
Single drive, double sided
Drive 2
Drive Ø
Drive 1
Double drive, double sided
Drive 2
Drive 3

Fig. 1.4. How drives are numbered. A single drive is always Drive Ø.

Storage space.

How much can you store on a disk? If we take a 40- track disk, then with ten sectors per track we will have 400 sectors. Each of these will store 256 bytes, which is a quarter of a kilobyte. Multiply 400 by a quarter, and you end up with a figure of 100K on a single-sided 40-track drive. Now this is very much on the optimistic side. For one thing, it ignores the amount of the disk that is used for storing information on the stored data, the catalogue. This occupies several sectors on the first track – a different number is used by the different systems. The Acorn system uses the first two sectors only. In addition, though, data is not stored at every point on the disk. Suppose you have a program that is 1027 bytes long. The disk operating system will split this into groups of 256 bytes, because it can record 256 bytes on one sector. When you divide 1027 by 256, you get 4 and a fraction – but the DFS does not deal with fractions of a sector. Five sectors will be used, even though the last sector has only three of its 256 bytes recorded. When the next program is saved, it will start at the next sector, so that the unused bytes are surrounded, and there is no simple way of making use of them. If you save a lot of short programs on the disk, you will find that a lot of space may be wasted in this way. A more serious limitation is that the Acorn DFS permits only 31 programs, or data files, per disk side. This is because only two sectors are used for the catalogue information. It can mean that you run out of catalogue space long before you run out of memory space on the main part of the disk, so that you get the 'DISK FULL' error message when the disk is, in fact, only half full. It is a special feature of the other alternative systems that they can both allow a large number of catalogue entries. Don't think, then, that you can place 100K of program or text on to a single-sided 40-track disk – you can't!

Access speed and the switch options

Disk drives made by different manufacturers will all carry out the same operations, but not necessarily at the same speed. It takes time to spin a disk up to the correct operating speed, and it takes time to place the head on the correct track. Different drives vary tremendously in these respects. For example, if we measure time in hundredths of a second, the time for a drive to settle to its set speed can vary from 16 to 50 hundredths of a second. The time needed to step from one track to another can range from 4 to 24 hundredths of a second. The disk operating circuits cannot cope with these differences, and the only way that this can be done is by making or breaking a set of links which are placed at the front of the keyboard (inside the case). Figure 1.5 shows how these links are arranged, and

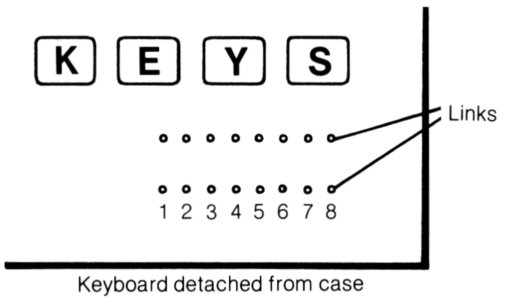


Fig. 1.5. The keyboard links. These are at the front right-hand side of the keyboard, and can only be seen when the cover is removed. Unless you have had some experience in taking your computer apart, it's best to leave them as they are!

Fig. 1.6 shows how they are used. The PACE/AMCOM complete disk kit comes with a switch which can be soldered into these holes,

(a) Use of links

Link	Acorn and Watford use	PACE use

1	Unused	Made=80track
		Unmade=40-track

2	Unused	Made=SYS1 at switch-on or BREAK
		Unmade=SYS∅ at switch-on or BREAK

3	Access time	Access time
4	Access time	Access time

5	Made=Boot on BREAK
	Unmade=Boot on SHIFT/BREAK
		Made=Boot on BREAK
		Unmade=Boot on SHIFT/BREAK

6	Decide screen	As Acorn Mode
7	Mode at	
8	Switch-on	

(b) Typical access time settings

Drive	Link 3	Link 4
Olivetti (Acorn)	Open	Open
MPI	Open	Closed
Shugart	Closed	Open
Tandon	Closed	Closed
Canon (new)	Closed	Open

Note: Old Canon drives need Olivetti settings.

(c) Mode settings

Screen Mode	Link 6	Link 7	Link 8
7	Open	Open	Open
6	Open	Open	Closed
5	Open	Closed	Open
4	Open	Closed	Closed
3	Closed	Open	Open
2	Closed	Open	Closed
1	Closed	Closed	Open
∅	Closed	Closed	Closed

Fig. 1.6. (a) The uses of the links for Acorn, Watford and PACE/AMCOM systems. *(b)* Typical access time settings for some disk drive systems. *(c)* Screen Mode at switch-on.

so that the links can be made or broken at will. The important point is that if you leave all of the links as they are, not connected, you can use any normal drive. This is the slowest setting, so that even the slowest drives have time to operate. If you have fast acting drives, like the Tandon or Shugart drives, you can, with advantage, close some of the links (or put some switches on) to make the system work faster. Do not, however, attempt to make your drives operate faster than they were intended to. If you try, you will find that the catalogue will record, and some of the early sectors will too. No recording, however, will take place on the later sectors, and you will get 'Faulty disk' or 'Faulty drive' error messages. It's wise to treat any information on access times with suspicion unless it comes straight from the manufacturers of the drives. The link setting for Canon drives, for example, is applicable only to some models, the more recent ones. Canon drives purchased before June 1983 should be operated with no links closed. The rule is simple – if you aren't sure, leave it alone!

Finally, Fig. 1.7. lists some precautions on the care of disks. These may look rather restrictive, but remember that a disk is precious. It can contain a lot of data, perhaps all of your programs. An accident to one disk, then, can wipe out all your work at the keyboard! Always make a back-up copy, and always take good care of your disks.

Care of disks

1. Don't bend the disks. They may be called 'floppy disks', but the magnetic coating is liable to be damaged if you bend them.
2. Always buy disks with hub reinforcing rings. The mechanism that clamps the disks in place will soon tear the centres of unreinforced disks. If you have any such disks with programs on them, make back-up copies on to disks with reinforcements.
3. Avoid touching the magnetic surface where the head slot is placed. Never try to take a disk out of its casing.
4. Store your disks, in their envelopes, in a box. You can keep them in the boxes (for ten disks) in which they arrive, or in boxes made for the purpose. Keep them away from dust, smoke, liquids, heat and sunlight.
5. Avoid, at all costs, magnets, or objects that contain magnets. These include electric motors, shavers (not many people shave while they are computing, but you never know), TV receivers, and monitors, telephones, tape erasers, electric typewriters, and a host of other things you might be tempted to place disks on.
6. Don't use a ball point pen to write on to labels on the disk. BEROL make a 'floppy-disk pen' which has a point that will break off if you exert too much force. A felt-tip is suitable, but you must not press hard while writing.
7. Use disks that are suitable for the drive you have. Do not use double-sided disks on a single-sided drive.

Fig. 1.7. Caring for your disks. This list makes disks look delicate, but is intended to ensure a long useful life. I have, for example, never had to scrap a disk. I have one disc which is slightly too large to spin correctly in one drive, but which operates perfectly in the other drive.

Chapter Two
The Disk Filing Systems

What does the DFS do?

The Disk Filing System, or DFS is, as we have seen, a program. This program is not written in BASIC, but in the form of direct commands in number code to the microprocessor (the 6502) which operates the BBC machine. Code of this kind is called *machine code*. If you want, or need, to know more about machine code, then I suggest that you turn to A. P. Stephenson's book *Discovering BBC Micro Machine Code*, also published by Granada. The purpose of the DFS is to interpret the disk commands that you type, and convert these into signals that can be used to control the disk system and shift data to and from it.

Note that the name is Disk *Filing* System, not simply Disk System. Filing implies the storage of data (such as string or number arrays) as well as BASIC or machine code programs. The DFS is therefore equipped to carry out the organisation of data which is needed to store it on disk and recover it later. That's something that we'll come back to later, in Chapter 8. Meantime, we'll keep to the more straightforward uses of the DFS. Rather than looking at the commands of the DFS in alphabetical order, we'll look at them in the order that is most likely to be of use to you, starting with the formatting of disks and their use for storing programs. First, however, we need to look at how the use of a DFS modifies the BBC machine, and what problems this can create for you.

The different systems

To start with, we need to look at some of the facilities that are offered by the Acorn, AMCOM, and Watford DFS chips. It's important to note that the alternative DFS chips offer all the facilities that the Acorn DFS has, but have additional advantages. Whether these advantages are sufficiently attractive for you to use one of these systems is a decision that you must make for yourself. If you are carrying out your own upgrade to a disk system, the decision is easier. If, however, you have bought a complete disk system machine, with all chips installed, then the decision to buy a new DFS chip and replace the Acorn one is much more difficult. The factors that you will have to think of are listed in Fig.2.1. For some users, these extras may not appear to be worth the expense and bother of replacing the DFS. For other users, they may make all the difference

Acorn: A standard system which will be supplied with a disk machine. It uses 2817 bytes of RAM. It allows 31 filenames in the catalogue. Program titles must not be more than 7 characters long. It requires a separate 'utility' disk (supplied with an Acorn disk drive) for formatting.

Watford: An optional system, supplied either as a DFS chip, or as part of a complete disk upgrade by Watford. It will work to Acorn standards, or optionally with 62 filenames and with additional commands. It requires the same amount of memory space as the Acorn system. It will format disks without the use of a 'utility' disk. Contains tape-to-disk transfer commands.

PACE/AMCOM: An optional system, supplied either as a DFS chip, or as part of a complete upgrade, which includes a switch for the keyboard links (soldering is needed for this). It will work to Acorn standard, or optionally with 62 filenames, and with longer filenames. It requires only 1793 bytes of RAM and will format disks without the use of a separate 'utility' disk.

Fig. 2.1. Comparing the different systems. This is a very brief guide only, and the differences will be explored in much more detail in the course of this book.

between a system which is simple to use and one which is not. There is no perfect answer. It is important to emphasise, however, that the alternative systems still permit you to save and load disks in the standard Acorn form. Disks which you record can be read by users with the Acorn DFS, and conversely, disks made on standard DFS machines can be read on your system. Without this compatibility, these alternative systems would be very much less attractive.

Losing your memory

Memory is one of the vital statistics of a computer, and it is organised in units that are called *bytes*. Each byte can store one character, but numbers are coded so as to make more efficient use of memory than having just one byte allocated for each digit. The total amount of memory that the microprocessor of the machine can cope with in one lump is 65536 bytes. So as to distinguish one byte from another, we number them, starting with ∅ and going up to 65535 in our ordinary counting scale.

Now different parts of the memory are used for different actions. Pages 500 and 501 of the BBC User Guide show how some of the memory is allocated. The most interesting feature for us at the moment is that memory addresses ∅ to 3583 are allocated for operating system use. This means that when we write a BASIC program, we start using memory at address 3584 upwards. This address of 3584 is called PAGE, and when you type PRINT PAGE, and press RETURN, you will see the number appear. PAGE, however, is a quantity that can be varied, and that's one feature of using a DFS that we need to look at now.

When you install a disk system, complete with DFS chip, or when you switch on a BBC machine which has a disk system installed, the first point you need to know about is the loss of RAM memory. A DFS chip is put in to organise data so as to make effective use of the disk, and it cannot do this without using RAM. This memory is taken from the lower end of the available RAM. On the cassette machine, the memory starts at address 3584, as we have noted. When the Acorn DFS is used, this number (PAGE) is changed to 64∅∅. This means that you have lost the use of 2817 bytes of memory. It's not very much if you use comparatively short programs, but it can mean that a long program on a cassette cannot be loaded! We'll look at that problem again later. Meantime, it's worth noting that the PACE/AMCOM DFS sets PAGE at 5376, losing you only 1793 bytes. This alone may be a useful factor in deciding which DFS to use. The Watford DFS uses the same setting of PAGE (at 64∅∅) as the Acorn DFS. Because the setting of PAGE can be changed, however, it is still possible to make use of all but the longest programs when you change from cassette to disk, though the longer the program is, the more complicated are the methods. You'll notice that the User Manual, and most other books and articles on disk operation also, use a different way of indicating these numbers. This is *hexadecimal* (or *hex*) code, and its use is so prevalent in any printed information on disk operating systems, that you need to know about it to get the most from your disk system. For the moment, though, we'll confine ourselves to the most straightforward use of the DFS – to place on disk BASIC programs that were formerly held on cassette.

Formatting a disk

Disk use has to start with formatting some new disks. With the Acorn DFS, this requires you to load the formatting program from the utility disk, and then place a new disk in the drive (assuming a single drive). Assuming that you have disk drives that grip the disk horizontally, hold the disk in your hand as the sketch of Fig. 2.2. illustrates. The side carrying the label (on a single sided disk) is

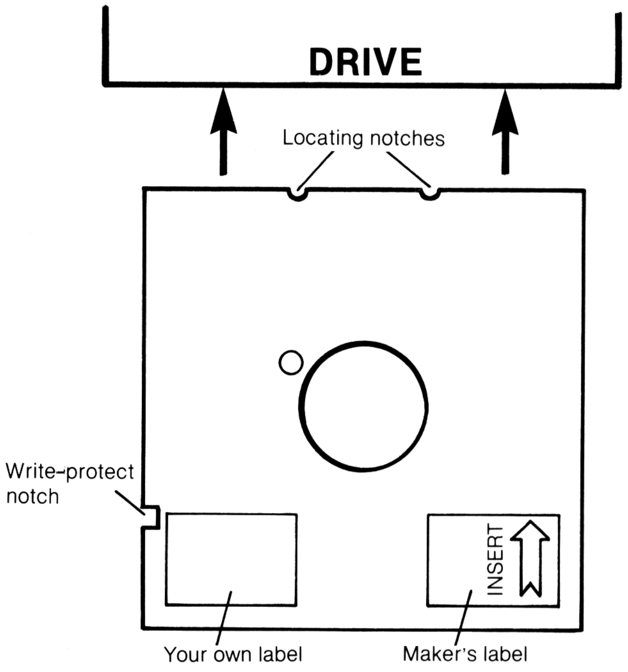


Fig. 2.2. Inserting a disk. Disks are a tight fit into the drives, and it's easy to jam a disk if you hold it slightly squint. Don't use force!

uppermost, and the long slot that allows the head of the disk drive to touch the disk is held away from you. Most types of disks carry a reminder in the form of an arrow on the label to jog your memory on this point. Don't push the disk hard. Slip it into the drive, making sure that you keep it straight. The disk is a close fit in the drive, and if you hold the disk slightly squint, it will jam. When it's pushed as far into the drive as it will go, shut the door of the drive on it until the door latches shut, or until you can close the door lever, according to the type of drive you are using. You will then, if you are using a new disk, have to use *FORM4Ø or *FORM8Ø according to the drive you are using. *Never* try to use the *FORM8Ø on a 40-track disk drive. As I explained in Chapter 1, this can cause severe damage to some types of drive. As the disk spins and is formatted, the track numbers will appear on the screen, and as each track is tested, the absence of a question mark after the number indicates that the track is a good one. The track numbers are shown in hexadecimal, not in ordinary scale-of-ten, but this need not concern you, since it's only a report on how good the tracks are.

If you are using the AMCOM or Watford DFS systems, then formatting does *not* require the use of a utility disk. Both require you to type *ENABLE as a precaution against wiping out data on a disk that is not a new one. The PACE/AMCOM system requires you to make a choice, before you format, of which system you want to use. If you want to use the Acorn system, you need do nothing unless you have altered one of the links in the keyboard. If you want to make use of the special features of the PACE/AMCOM DFS, then you have to type *SYS1 and then press RETURN. This means that a disk will either be to Acorn standards or to AMCOM standards – you cannot choose to have part of a disk in one system and part in another. You then have to choose the number of tracks on the disk. Once again, if you have not altered any of the links on the keyboard, this will automatically be set to 4Ø. You can alter it to 8Ø in two ways. One is by having connected link No.1 on the keyboard. The other way is by typing *OPT3,8Ø (then RETURN). If you have set the links on the keyboard to suit your own drive and the system that you want to use, you don't have to do any more than type *FORMAT, then press RETURN. If you have more than one drive, then you will have to specify the drive number as, for example, *FORMATØ, or *FORMAT1. When the RETURN key is pressed, the formatting will be carried out, reporting on each *sector*. Each sector number is printed, in hex, and – when it has been tested – the letter G is printed to indicate a good sector. B would indicate a bad sector, and so an unusable disk.

The Watford DFS formats in a rather different way. After using *ENABLE, you need to type *FORM4Ø or *FORM8Ø, according to how many tracks your drive uses. If you have more than one drive, you can add a space and then the drive number such as, for example, *FORM4Ø 1. You are then asked if you want to use the larger catalogue that this DFS permits, by the question:

Large (62 file) catalogue? press Y

If you press Y in response to this prompt, the disk will be formatted to Watford standard, and will not be wholly compatible with an Acorn disk. If you make any other reply, the disk will be formatted to Acorn standard. As a further check of your intentions, you are then asked:

CONFIRM FORMATTING OF DRIVE ∅? – press Y again

(using whatever drive number you have specified if you have more than one drive). Pressing Y will now result in formatting being carried out. The message:

Formatting Drive ∅

(or whatever number you used) appears, and the track numbers then start to appear. If any sector is bad, then a question mark will appear following its number. If this happens, you should try formatting again. If two question marks appear, then it's likely that the disk is faulty – but make sure first that you are not trying to run your disk drive too fast by incorrect settings of the drive-speed links on the keyboard.

Loading and saving

Once a disk has been formatted, you can use it for storage. The method that you follow for BASIC programs is identical, no matter which DFS you are using. Load in the program that you want to save. If this program is on cassette, you will have to start by typing *TAPE (then RETURN). If you do not do this, then the machine will *always* use the disk system rather than the tape system. Unless the program is a very long one, loading from tape should present no problem other than the time it takes. For the problems associated with very long BASIC programs, see later in this chapter. Once the program has loaded, type *DISK (RETURN) to return to the disk system from the tape system. Then make sure that you have a formatted disk in the drive, correct way up. Type SAVE "MYPROG", using whatever filename you have decided to give the program. Remember that the Acorn and Watford systems permit up to seven character filenames only, but the AMCOM system allows up to fifteen characters. The inverted commas are essential when

LOAD and SAVE are used, but not when you use any of the *Commands because the DFS is programmed to look for the space after a *Command and make use of the word following it as a filename. You *must* use a filename when you load or save using disks. When you press RETURN, the disk drive will click, and almost immediately (unless it is a very long program) you will see the > prompt reappear to indicate that the transfer is complete. Shortly after this, the disk drive light (if there is one) will go out, and you will hear the disk motor stop. That's it! If you are using the PACE/AMCOM system, and you have formatted a disk in the extended mode (using *SYS1), then you cannot save a program on it unless SYS1 is still in use. If you have pressed BREAK, or switched off, and the keyboard links are not set so as to start up in SYS1, then you start in SYSØ. If you want to save a program on a disk so that it can be read by a machine equipped with the Acorn DFS, then you have to re-format, using SYSØ, and then save the program. This re-formatting can, however, be carried out while the program is in the memory, since formatting on the PACE/AMCOM system does not make such demands on memory as the disk-based program used by the Acorn system. If you try to SAVE or LOAD, using SYSØ, when the disk is formatted to SYS1, the screen will print an error message, and no harm is done. The same applies if you are using SYS1 and the disk has been formatted to SYSØ. The Watford DFS specifies its system only on the formatted disk – if you have formatted for a large catalogue, you will have a disk that cannot be used by the normal Acorn DFS. No action is needed at switch-on or after pressing BREAK.

To load a program that is on disk, you can type LOAD "MYPROG" (or whatever filename you have chosen) and press RETURN. If the disk is correctly inserted in the drive, it will spin, and the prompt will reappear shortly to indicate that the program is loaded and ready. If you used the wrong filename, you will either get the wrong program or an error message, depending on whether the file of that name existed. If you want to load and run, you can use CHAIN "MYPROG" in the same way as you use for cassettes. Loading is generally much faster than storing, because the DFS carries out a check on data when it records, but not to the same extent when it replays. If you get any sort of error message when you are saving a program, then it's wise to assume that the program has not been saved, and to save it again. Once again, if you had saved the program using the AMCOM SYS1, and have then switched off the machine, you may (depending on the keyboard links) start up in

SYSØ. If you have forgotten to change over, you will be reminded by a screen message (Bad system).

When you have saved a program on a disk, it's time to take a look at the way the disk keeps track of your program. Type *CAT to show this. If you have only one drive, that's all you need, but with two or more drives, you will have to select the drive number first by using *DRIVEØ, or *DRIVE1, etc., beforehand. An alternative is to use *CATØ, *CAT1, etc. Both of these commands are used by all three systems. When you press RETURN, the disk will spin briefly, and display the information that is illustrated in Fig. 2.3. This shows the title of the disk (if any), the drive number, the option (what happens when BREAK is pressed), the directory, the library, and (for the Watford system) the work-file. The PACE/AMCOM DFS gives a rather different *CAT display, but it conveys essentially the same information. When you first start to make use of disks, you don't need much of this, but you will probably find it more useful later. At first, the most useful feature of the catalogue is the fact that the program filenames are shown. The Acorn and Watford DFS systems show the files in alphabetical order, the AMCOM DFS shows them in order of storage on the disk.

(a) Acorn catalogue

```
UTILITYDISK
Drive: Ø              Option: 2          (RUN)
Directory: Ø.$                           Library :Ø.$

!BOOT                                    ANALYSE
CONTROL                                  DISASSR
EPSON1                                   INDEX
SECTOR                                   VARPRNT
```

(b) PACE/AMCOM catalogue

```
UTILITYDISK
Drive Ø               Option 2           System Ext
Directory $           Library Ø.$
!BOOT                                    CONTROL
VARIABLEPRINT                            INDEX
ANALYZEDISK                              EPSON1DUMP
SECTOREDIT                               DISASSEMBLER
```

(c) Watford catalogue

```
UTILITYDISK
Drive Ø                 Option 3            (EXEC)
Directory :Ø.$                               Library :Ø.$
Work file $.UTILITY

!BOOT                                        ANALYSE
CONTROL                                      DISASSR
EPSON1                                       INDEX
SECTOR                                       VARPRNT
```

Fig. 2.3. The typical catalogue displays that you can expect to see using the different systems: *(a)* Acorn *(b)* PACE/AMCOM *(c)* Watford.

Extended filenames, or file specifications

One feature of the DFS that is seldom of interest when you first start to use disks is the use of extended filenames. When you type a filename and use it to SAVE a program, what is actually passed to the disk is more than the filename itself. The other pieces of information are the directory name and the drive number. If you have only one drive, then the drive number will always be Ø, and that's that. If you have more than one drive, however, you can include the drive number as part of your filename, so as to cause your program to be recorded on a disk in that specified drive. When you want to do this, you have to use the colon symbol (:) to indicate to the DFS that this is an extended filename. For example:

```
SAVE ":1.VARNAM"
```

would save a program on to the disk in Drive 1, using the filename of VARNAM. This is a shorter way of carrying out the steps of *DRIVE1 (RETURN) followed by SAVE "VARNAM".

The other extension to the filename is to specify the directory. Directory in this sense means an identifying letter which can be used so as to group files of the same type. If you save only BASIC programs on a disk, it isn't very important to use this facility. Suppose, however, that you had some BASIC programs, some machine code programs, and some number arrays (the results of your Accounts program, perhaps) stored on a disk. The names of the files alone may not be enough, after a few moons have waxed and waned, to remind you which file was which. This is the use of the directory letter. Suppose you used B to mean BASIC, M to mean machine code, and X to mean your number arrays. If you have a BASIC program called "RAYDIAG", and you want to save it on Drive 1, using the B directory, then you type:

SAVE ":1.B.RAYDIAG"

to do so. This is what is meant by a 'full file specification'. If you have only one drive, Drive Ø, then the number can be omitted, so that you can use SAVE ".B.RAYDIAG" to attach the directory letter of B to this file. If you treat all the files on a disk in this way, using the different directory letters according to the type of file, then these directory letters are displayed when *CAT is used, as Fig. 2.4 shows. The first directory that is shown is the 'current directory',

```
WORKDISK1
Drive:Ø            Option: 3          (EXEC)
Directory: Ø.$                        Library :Ø.$
!BOOT                                 FIRST

B. ADDIT                              M. DISSASSR
Z. ASSMAID
```

Fig. 2.4. A typical catalogue for a disk that contains several different directory letters.

meaning the one that is being used at the time of typing *CAT. A useful point to note is that files which have the same name, but different directory letters, are regarded as different. Saving a file called ".B.MYPROG" will not erase a file called ".A.MYPROG" on the same disk, for example. The files are shown in alphabetical order in their respective directories. If you do not specify any directory, then the symbol $ is automatically used by the DFS. This is the 'default directory', just as Drive Ø is the default drive. When you first start to make use of disks, you can omit the extensions to the filenames, and simply use the default Drive number and Directory letter.

The Watford system permits the addition of a 'workfile'. A *WORK command (followed by drive number and filename) can be carried out. This will automatically ensure that if you use no other filename, the filename specified in the *WORK command will be used. Suppose, for example, you typed *WORK :1.A. TEST. This would ensure that the command SAVE "" would save a file to Drive 1, using Directory letter A and the filename TEST. It is an aid to quick saving and loading. The workfile directory and name appear when you use *CAT using the Watford DFS.

Titling the disk

The directory facility is useful if you want to keep different types of files on one disk. As your use of disks increases, however, you may find that you want to group files that are related in some way on to one disk. It is then very helpful if you can give this disk a title which will remind you of what it contains. You might, for example, have a disk full of utility programs of various types, and a logical title would be UTILITIES. You can use a title name of up to twelve characters. The Acorn and Watford systems allow this title name to contain teletext control codes, so that the titles can appear in colour. The PACE/AMCOM system does not permit this. In addition, Acorn disks which use coloured titles can cause the AMCOM system to attempt to read the disk as if it had been created by the SYS1 option. The AMCOM chip comes with a utility disk which contains a program called ACORNISER to correct this problem.

You can title a disk by using the *TITLE command. This has to be followed by a space, then the title (12 characters maximum) that you want to use. This title will then appear at the top of all the catalogue information, reminding you of what you have used the disk for. You can also change the default directory, if you want all the files that will be put on the drive to have the same directory letter, and you don't want to use the $ character. To alter the default directory, you type *DIR, followed by the drive number (optional) and the new directory letter. You cannot use the characters # * . or : as directory characters, because they have special uses. For example, you could use *DIR: 1.A to make the current directory letter A. For as long as you have the machine switched on, then, all files will be saved to this directory if you don't specify any other. Previously recorded files will retain whatever directory letter was used when they were saved. If, however, you use a store command such as STORE ":1.B.INDEX", then this file will be saved on Drive 1 with the directory letter B, no matter to what you have set the 'default directory'. As usual, if you have only one drive, you don't need to use a drive number. If you have more than one drive, and omit the drive number, then Drive ∅ will be used.

Dealing with long programs

One of the most awkward problems that you are likely to encounter in the process of transferring your BASIC programs from cassette to disk is how to deal with very long programs. Programs which will load from tape and run normally in a disk machine are no problem – you simply select *TAPE to load from cassette, and *DISK to save on disk. As we have seen, the use of a DFS grabs some of the RAM memory which would otherwise be used for programs. It is possible, therefore, for a program which could be loaded into the machine previously to be too long to fit when a disk system has been installed. Let's start with the simplest option – you have a very long program on tape which you just want to RUN, not to copy to disk. You first have to restore the cassette operating system by typing *TAPE (then RETURN). The next step is to restore the normal starting place in memory. This will alter all of the memory that has been used by the disk system, but if you don't want to do anything other than run the program, it doesn't matter. To do this, type PAGE=3584 and press RETURN. You can then use the machine just as if you had never fitted a disk system. The only proviso is that LOAD and SAVE will now be to the cassette system, and commands like *CAT will not operate the disks. When you use BREAK, or switch off and on again, the DFS will take command again, and the PAGE setting will go back to the number that is used by the disk system, so that you will have to repeat the *TAPE and PAGE=3584 steps if you have done either of these things. Note the different setting of PAGE for the AMCOM system.

Things get more difficult if you want to transfer a program from tape to disk, when the program is too long to fit in the memory along with the DFS. As it happens, the amount of extra RAM which is used by the DFS is not necessarily used for all actions. If you use nothing more than the ordinary LOAD and SAVE (or *LOAD and *SAVE) commands, the memory above address number 46ø8 (Acorn and Watford), or 4352 (AMCOM) is not used by the DFS. You can therefore shift PAGE to this value and still use these important commands. This does not solve the problem as far as BASIC programs are concerned, however. This is because the PAGE value will have been reset when the program is loaded from disk once again, after a BREAK or after having switched off and on again. This can be dealt with by using a short program to set the PAGE value, and then CHAIN the main program. For example, if you use:

```
10 PAGE =4608:CHAIN"TOOLONG"
```

then this line can be saved with the filename that you want to use for the main program, and the main program can be saved with the filename of TOOLONG. When you CHAIN the short program, it will set the start of memory down, and then load in the main program and run it. The main problem now is to ensure that the program on the tape is transferred to disk correctly so as to make use of this method of running. The procedure is as follows:

1. Use *TAPE to switch to the cassette system.
2. Use *LOAD""1200 to load in the program, with the tape placed at the start of the BASIC program. Note that this is * LOAD, not LOAD. The effect of the command is to treat the program as if it were a machine code program, and load it into memory starting at the hex address 1200, which is 4608 in ordinary numbers. *LOAD 1100 can be used with the AMCOM system.
3. When the tape has loaded, note the second in the pair of numbers, which is the length of the program. This again is a hex number. Do not LIST or RUN the program.
4. Use *DISK to reselect the disk operating system.
5. Use *SAVE "name" 1200 XXXX to save the program on disk, with a filename that you have to supply. Users of the AMCOM system can use the number 1100. The problem here is to calculate XXXX. This is a finishing address for the program, and it is obtained by adding the length number to the starting address of 1200 or 1100 (hex). The snag is that both of these numbers are in hex, so you need to use the machine itself to help you add them if you are not proficient in hex arithmetic. Suppose, for example, that the length number comes up as 0FB6, and that the AMCOM system is in use. Using direct mode, type:

   ```
   PRINT ~ (&1100 + &0FB6)
   ```

   and then press RETURN. The screen will show the result of the addition in hex, which is what is needed for the *SAVE command. If you are not confident about this, then postpone attempts to use this method until you have looked at the section on hexadecimal numbers in the next chapter. An alternative is to use *SAVE 1100 +0FB6, where the + sign causes the computer to calculate the end address for itself.

Once the program is recorded on disk in this way, it can be entered and run by using the CHAIN method that was discussed earlier. Alternatively, you can use PAGE=&12ØØ (the & means that the number is in hex), followed by LOAD "name", if you want to list before running, or to make changes.

There may be a few commercial programs which will not fit on to disk in this way. If so, there are methods of dealing with some of them, but they are beyond the scope of this book. Commercially available programs exist for transferring the 'awkward' tape programs on to disk, and these are useful for all but a few examples. A program which simply cannot be transferred will just have to be left on tape until a disk version is available. It would be pleasant, though optimistic, to think that the suppliers of such programs might make an allowance on the price of the disk version if the cassette version is returned undamaged.

The problem is very much simpler if you are dealing with a program of your own. The fact that one program can CHAIN another rapidly from disk means that many programs can be split into portions. The problem then becomes one of passing variable values from one program to another. Several methods of doing this have been published, all involving manipulation of the 'variables map' in the BBC machine. A much simpler alternative is to save all the variable values as data files (see Chapter 7). These can then be read by the second part of the program. The speed of loading and saving on disk is such that there is no disadvantage to using this type of method.

Chapter Three
Digging Deeper

Hexadecimal codes

Unless you program in machine code, you probably haven't encountered the *hexadecimal scale*. 'Hexadecimal' means scale of sixteen, and it's a way of writing numbers that is much better suited to the way that the computer uses number codes. Our ordinary number scale is *denary*, scale of ten. This means that we count numbers up to nine, and the next higher number is shown as two digits, 1∅, meaning one ten and no units. Similarly, 123 means one hundred, two tens and three units. This counting scale, invented by the Arabs, replaced the Roman numbering system many centuries ago (except, oddly enough, for writing the dates of films and TV programs!). A denary number for a byte may be one figure (like 4) or two (like 17) or three (like 143). Hex (short for hexadecimal) is a much more convenient code for these numbers, and for address numbers. All one-byte numbers can be represented by just two hex digits, and any address by four hex digits.

One hex digit, then, can represent a number which, written in ordinary denary, can be between ∅ and 15. Since we don't have symbols for digits higher than 9, we have to use the letters A,B,C,D,E and F to supplement the digits ∅ to 9 in the hex scale, as Fig.3.1 illustrates. The advantage of using hex is that we can see much better how address numbers are related. For example, consider the values for PAGE that we have been looking at. The values of 3584, 5376, and 64∅∅ in denary become hex numbers ∅E∅∅, 15∅∅ and 19∅∅. Their common factor is that the last two digits are ∅∅. Any change of the PAGE value *must* be to a value which has both of its lower digits zeros. Thus we can use hex 21∅∅, which is denary 8448, but not hex 2328, which would be denary 9∅∅∅. Let's take a formal look at what this scale is about.

Denary	Hex	Denary	Hex
1	∅1	9	∅9
2	∅2	1∅	∅A
3	∅3	11	∅B
4	∅4	12	∅C
5	∅5	13	∅D
6	∅6	14	∅E
7	∅7	15	∅F
8	∅8	16	1∅

Fig. 3.1. Denary and hex digits. The hex scale uses the letters A to F as symbols for the numbers ten to fifteen.

The hex scale

The hexadecimal scale consists of sixteen digits, starting as usual with ∅ and going up in the usual way to 9. The next figure is not 1∅, however, because this would mean one sixteen and no units. Since we aren't provided with symbols for digits beyond 9, we use the letters A to F instead. The number that we write as 1∅ (ten) in denary is written as ∅A in hex, eleven as ∅B, twelve as ∅C and so on up to fifteen, which is ∅F. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four even if fewer digits are needed. The number that follows ∅F is 1∅, sixteen in denary, and the scale then repeats to 1F, thirty-one, which is followed by 2∅. The maximum size of byte, 255 in denary, is FF in hex. The maximum size of address in the memory of the computer, 65535, is hex FFFF.

When we write hex numbers, it's usual to mark them in some way so that they are not confused with denary numbers. There's not much chance of confusing a number like 3E with a denary number, but a number like 26 might be hex or denary. The convention that is followed by users of the BBC machine is to mark a hex number with the ampersand sign (&) placed before the number. For example, the number &47 means hex 47, but plain 47 would mean denary forty-seven. The machine itself will recognise the use of & to mark a hex number and convert it to denary, so that you can enter numbers like &∅E∅∅ or &15∅∅ for PAGE values which must normally be in denary. The BBC machine contains routines for the conversion of numbers between hex and denary scales, so that you never need to carry out hex arithmetic for yourself.

If you type PRINT &1234, for example, the machine will show the number 466∅. This is the denary equivalent of &1234. To convert denary into hex, the tilde sign (~) is used. This is on your keyboard above the 'inverted V' sign, next to the = key. When you are in Mode 7 graphics, as you would normally be when programming, this tilde sign appears on the screen as a division sign, but it will print correctly on most printers. The instruction PRINT ~1234 will cause the screen to show 4D2 as this hex number. We would normally write this as &∅4D2. To find the hex result of adding two hex numbers, then, as a previous example did, you need to use the & sign on the numbers, within brackets, and have the tilde sign outside the first bracket. For example, PRINT~ (&1234+&1CF2) will give the result 2F26. From now on, then, I'll use hex numbers where they are more suited for address numbers (sometimes called 'words') or data bytes. Note that you have to be careful not to use the & sign with a hex number for some commands that require the hex number only. The *LOAD and *SAVE commands (see later) are of this type.

Backing up

One feature of a disk storage system which is less pleasant is that an accident to a disk can result in the loss of a lot of information. If you break a cassette tape, it's possible to splice the tape, and with some juggling, lose only a part of one program. If you damage a disk, it's likely that all of the information on the disk will be lost as far as conventional LOAD commands are concerned. As we'll see in Chapter 6, this does not mean that the information cannot be recovered from the disk, but this is a desperate measure, not to be undertaken lightly. It makes sense, then, if you have a disk full of valuable programs or data, to make a back-up copy as soon as possible.

One sensible measure is to make a second copy of each program as you put it on disk. If you have used one of the commercial programs (or one of the published listings) which automatically places programs from a cassette on to a disk, however, you may end up with a disk which has several programs on it, and no copy. The DFS copes with this by allowing a *BACKUP command. Now for BASIC programs this does not do anything that you could not do for yourself by typing LOAD"PROG1" followed, when the program was in place, by swopping disks and typing SAVE "PROG1". In other words, the use of *BACKUP will read each program (or other file) into the memory of the computer from one disk, and write it out to another. The important point is that the memory of the computer will be overwritten when this happens, so any program that you have had in the memory will be destroyed by this process. Because of this, *BACKUP must always be preceded by *ENABLE to make you think twice about using this facility. The great advantage of *BACKUP, however, is that it transfers everything on one disk to another. This includes BASIC programs, machine code programs, data files, text files, the lot. All will be transferred and stored under their original filenames.

This is most convenient when you have twin drives. With two drives (*not* just a double-sided drive) in use, you can use *BACKUP ∅ 1 (RETURN) to cause everything on the disk in Drive ∅ to be copied to the disk in Drive 1. The process is accompanied by a lot of clicking and whirring, as one disk is read and the other written, but at least you don't have to attend to the process. You can make yourself a cup of coffee while it is all happening. The process is equally automatic if you have a double-sided drive (using *BACKUP ∅ 2 for example), but it's undesirable. The reason is that if you back up a disk on to the reverse side of the same disk, your back-up copy is subject to the same risks as your first copy! It's much safer to back-up on to another disk, and to keep this back-up disk in a cool safe place well away from all the hazards of disks, such as loudspeakers, TV receivers, electric motors and anything else that uses magnets of any kind.

If, like many first-time disk users, you have only one drive, don't worry, because you can still make a back-up. The procedure is to place the disk in the drive, type *ENABLE and then *BACKUP ∅ ∅. You will then get a series of screen messages asking you to place either the source (original) or the destination (back-up) disk in the drive and press RETURN, until backing up is complete. This process needs your attention, and is an excellent way of persuading you that twin drives (as the Gillette advertisement says) are better! Both the AMCOM and the Watford DFS systems use the same version of this command.

Copying a named file

Very often, you don't need to back up a complete disk, just copy one file on a disk to a back-up disk. The command which makes this possible is *COPY. This will also make use of the memory of the computer, so that any other program which you have in the memory will be overwritten. Unlike *BACKUP, however, *COPY does *not* require the use of *ENABLE, so that you get no reminder of the destructive effect of this command. The command *COPY has to be followed by the drive numbers, source drive then destination drive, and then the filename. For example, *COPY ∅ 1 DISCOVER will copy the program called DISCOVER from the disk in Drive ∅ to the disk in Drive 1. Once again, if you have only one drive, you will be prompted to insert the source disk, and then the destination disk in turn.

There are a few quirks to *COPY, however, that do not exist on *BACKUP. One is that *COPY makes efficient use of the destination disk. If a program has been deleted on the destination disk, and there is room for the program that is being copied, *COPY will insert the copy in the space. This does *not* happen if you simply SAVE a new program on to a disk which has had such a deleted space, as we'll see later. The other point is that a file with the same filename may already exist on the destination disk. If this is so, the copy action will not proceed, and you will see a 'File exists' message on the screen. This is a reminder that you are in danger of wiping out a file which you may have forgotten about. If you actually want to do this – as, for example, when you are updating a file and want to keep using the same name – you will have to delete or rename the old file before you use *COPY. We'll deal with both of these processes later in this chapter.

The PACE/AMCOM DFS allows you to use an ambiguous filename with *COPY. For example, *COPY ∅ 1 #W* will copy all files whose names start with W, regardless of directory letter. There is also a unique facility which applies to *COPY and a number of other commands. This allows you to prevent *COPY from overwriting a program in the memory. To make use of this, you have to know the highest address in the memory that the program occupies. You can find this by typing PRINT TOP. This will give a number (in denary) which is the highest address of your program (though higher address numbers are used when it runs).

Convert this number to hex, and round up to the next higher number which ends in ∅∅. Then use the first two digits of this number in an *OPT5,XX command. For example, suppose that you found TOP = 5375 in denary. Converting this to hex gives &14FF. The next higher number ending in ∅∅ is &15∅∅, and hex 15 is denary

21. You can therefore type *OPT5,21 (or *OPT5,&15) to protect your memory. If your program is a very long one, however, the result of this may be a 'No space' report. In this case you must SAVE your program before you attempt to use the *COPY (or other) facility. If, however, the use of *OPT5 is acceptable, you then need *OPT7. This allocates how much of the rest of the memory is going to be used. This one is needed because you may have a piece of machine code occupying the top of the memory. The figure that follows *OPT7 is a number of complete 'blocks' of memory. One block for this purpose is the same amount as is stored on one sector of a disk, 256 bytes. If, for example, you want to use only 4Ø96 bytes of memory, then this is 16 blocks (because 4Ø96/256=16), and *OPT7,16 will do what is needed. To find how much memory is available, subtract the address that you have used in *OPT5 from the highest address that you want to use. If you are not using any part of the top end of memory, this address will be given by typing PRINT HIMEM. Figure 3.2 illustrates this use of *OPT5 and *OPT7.

```
PRINT~TOP
3614
```

For example:

Take first two (hex) figures and round up: 37.
Find denary equivalent by typing PRINT &37. This gives 55.
*OPT5,55 will protect the program.
If you are using Mode 6, and your screen display starts at address &6ØØØ, then you can use &37ØØ to &6ØØØ for copying or compacting. You can find what this is in denary by typing:

```
    PRINT &6ØØØ — &37ØØ
```

and this will give 1Ø496. Divide this by 256, and you will get 41. This is the number to use with *OPT7. The command is:

```
    *OPT7,41
```

This will use as much space as is available in *COPY, *COMPACT and *BACKUP operations.
Note: *OPT5 and *OPT7 use *denary* numbers.

Fig. 3.2. Protecting a program against the effects of commands which use memory. This can be done on the AMCOM system by using *OPT5 and *OPT7. This example shows the steps that are needed, starting with the use of PRINT~TOP to find the highest address in memory that is used. The figure of 3614 is a typical one.

Deleting files

As well as copying and creating files on to disks, you may want to delete files. You may, for example, have developed a BASIC program through several versions, and wish now to delete all the old versions. You may, to take another example, have an accounts program which creates a file of inputs and outputs of money, and which needs to read a data file in, and write one out to update the data. This also *may* require you to delete an old file – but we'll discuss data filing in more detail in Chapter 7. Whatever your need, deleting a single file is carried out using *DELETE. This command is available on Acorn, AMCOM and Watford filing systems alike.

*DELETE has to be followed by the filename of the file that you want to delete. This does *not* remove the data of the file from the disk. What it does is to remove the catalogue entry, so that the space on the disk can be used by a later entry. This will happen only if the new file is shorter than the deleted file, or of the same length. If this is not so, the new file will use another part of the disk, and the space that was used by the deleted file will remain unused until we do something about it. It is possible to recover the contents of a deleted file by writing a new catalogue entry, but this comes into the realms of advanced programming and is definitely not the sort of thing you want to attempt while you are getting to know your way round the DFS! The *DELETE action will not work on a disk that is 'write-protected' (see later, this chapter).

Wild-cards and wiping

*DELETE is a way of deleting one single named file on a disk. There are also commands which will remove more than one file. One of these is *WIPE, which is used on all three DFS. *WIPE has the effect of removing, one by one, from the catalogue a *group* of files – but to use this, you need to specify some feature that the group will have in common. This involves the use of the 'wild-card' characters. This romantic phrase means that certain characters can be used to indicate any character or grouping of characters. The two 'wild-card' characters of the DFS are * and #. Of these two, * is more useful. The * can be used to stand for any group of characters of any length. Suppose, for example, that you had files called TEXT1, TEXT2, TEXTTEST and TEST. If you wanted to wipe the first three of these files, but not the fourth, you could type:

```
    *WIPE TEXT* (then RETURN)
```

The wild-card character * means that whatever follows the second T of TEXT doesn't matter – any file that starts with TEXT is to be wiped. This can be rather dangerous if you are careless, so the *WIPE command gives you a second chance to save any file that might be valuable. When *WIPE is used, each filename is printed in full on the screen, along with a colon. The colon is a prompt for you. If you type Y, then the file will be deleted. If you type any other letter, the file will not be deleted, and another file will be selected from the catalogue and its name presented to you. If you had typed *WIPE T* rather than *WIPE TEXT*, then all the file names that started with T would be presented for your thumbs-down, not just the TEXT files.

The other wild-card character, #, is less useful. # will represent any *single* character, but not multiple characters. If, for example, you have a set of BASIC programs saved as VERS1, VERS2, VERS3, and so on up to VERS9, then typing *WIPE VERS# will present all of them for possible removal – but it would not affect VERS1Ø or VERS11 because these use two characters.

The wild-card characters can also be used with the *COPY command. For example, *COPY Ø 1 TEXT* will copy from Drive Ø to Drive 1 all files whose filenames start with TEXT. This is a very useful facility which can save a lot of typing. Typing is a process which is well-known for introducing errors, as my editor can testify. Using the wild-cards, you need type only the bare essentials of a file specification, and can then leave the machine to do the rest. It can also ensure that if one filename is misspelled, it will not be ignored on this account. A filename which uses a wild-card character is usually referred to as an ambiguous filename, and this is how we shall refer to it from now on.

Another of the file-removing commands, found on the Acorn and Watford DFS, is *DESTROY. *DESTROY can be followed by an ambiguous filename, and will, like *WIPE, remove from the catalogue any files that correspond to the specification. This one, unlike *WIPE, has to be preceded by *ENABLE. Like *WIPE, *DESTROY causes the filenames to be listed, and pressing Y will cause *all* of the files on the list to be deleted. Any other key allows a reprieve. Note that you don't get the chance to decide the fate of each file in turn, as you do with *WIPE. You have to decide whether or not to delete the whole *group*. The AMCOM DFS does not employ the *DESTROY command, but has the *CLEAR command which will delete all the files on a disk. This needs the use of *ENABLE.

A more drastic way of wiping a disk, if you want to remove all of the files, is simply to format it again. Remember, however, if you use the Acorn DFS, that formatting means using memory, and any program will be overwritten. If you use the other commands which remove files, you will be left with a disk which bears a strong resemblance to a piece of Emmenthal cheese. That means it's full of holes, unwanted bytes of data that are not used by any file that is in the current catalogue. We can make more efficient use of the disk by reallocating this space, so that all the files we actually have in the directory are put into the first parts of the disk, rather than scattered all over it. This requires the use of *COMPACT.

Every now and again, you will get a 'Disk full' message on a disk which you know should have plenty of space on it. This is because files have been deleted from the disk, but subsequently entered files have been too large to fill the gaps. The gaps therefore remain, preventing the addition of data. The disk *is* full, but not of wanted data! *COMPACT reallocates space on a disk by the simple method of reading files from the disk and writing them on again, using all the disk space in the lower-numbered sectors. It's therefore a command which will wipe out any program in the memory. *COMPACT can be followed by a Drive number. If this is omitted, it will compact the disk in whatever Drive is currently in use. As each file is rewritten on to the disk, information about it is printed on the screen. This is the same information as is given by the *INFO command, of which more will be said later. After all the files have been rewritten, the amount of free sectors is printed. You can find how many bytes of free space remain on the disk by multiplying this number by 256. Remember, though, that the 'free sectors' number is in hex, so that you have to prefix it with & before multiplying. For example, if you find the figure A8 after compacting, then you must type PRINT &A8*256 (RETURN) to find the number of free bytes, in denary. If no files have ever been deleted from a disk, then *COMPACT will simply cause a listing of the names of the files along with the number of free sectors. This makes it a useful way of deciding whether a disk can accept another file or not.

The Watford and the AMCOM systems allow some modifications to this scheme. The Watford system allows you to type *HELP SPACE before you compact a disk. This will list all the spaces that the *COMPACT command will remove. It's a very useful way of helping to decide whether a *COMPACT operation is needed. The AMCOM system does not use *HELP SPACE, but will allow you to carry out a *COMPACT without overwriting memory. This makes use of *OPT5 and *OPT7 as previously described. By using these commands, you can decide what range of addresses in memory will be used by the *COMPACT operation, avoiding the addresses in which your program is stored.

Protecting disks and programs

Just as we have a number of methods of deleting files from disks, we have also a number of methods of preventing this from happening. One method of doing this is universal to all disk systems on all machines. It makes use of the 'write-protect' tab on the disk. If you hold a disk as you would if you were inserting it into a horizontal drive, you will see a small rectangular slot cut from the left-hand side of the jacket. This is the 'write-protect notch'. When the disk is in the drive, the presence of this notch is detected, either mechanically or by a light beam. If the notch is unobstructed, the disk can be read and written. If the notch is covered, then the disk can only be read, not written again. In each pack of disks you will find a set of small sticky tabs that can be used to cover this notch to make a complete disk 'read-only'. If you want to reuse such a disk, you only have to remove the tab.

In addition to this method, which applies to all disks and all machines, the DFS allows you to protect files against all but a few commands. This is called 'locking', and the command which carries out locking (and unlocking) is *ACCESS. When a file has been protected using *ACCESS, it cannot be removed by *DELETE, *WIPE, *DESTROY, nor can its filename be changed by *RENAME (see later in this chapter). It cannot be overwritten by writing another file of the same name, and the only way that you can remove it without unlocking it is by reformatting the disk. Locking should therefore be used for any really vital file, if only to prevent overwriting by another file of the same name in a *BACKUP operation which uses the disks in the wrong drives. Note, however, that if a disk is copied by using *BACKUP, files that were locked on the source disk will *not* be locked on the destination disk.

*ACCESS has to be followed by the filename of the file that you want to work on. If the filename is followed by a space and then the letter L, the file of that name will be locked, with the effects that I have just described. If the letter L is omitted, the effect is to unlock a locked file. When the *CAT command is used to display a catalogue of files, locked files appear with a letter L following the filename. A file cannot be locked if the write-protection tab is fitted to the disk, because the action of *ACCESS means that some coding has to be added to the catalogue. If you want to lock a file on a write-protected disk, temporarily remove the tab until the locking has been carried out.

The PACE/AMCOM DFS permits a different approach to *ACCESS, in addition to the method that is also used by Acorn and Watford systems. When the AMCOM system is used in Acorn mode, SYSØ, the *ACCESS command works as just described. However, in SYS1 mode, another use of *ACCESS is possible. In this mode, the catalogue display will show only the filenames of locked files, not the fact that they are locked, nor their directory. This greatly extends the protection, because only the original user can then easily unlock the files. The only alteration that is required is the addition of a tilde sign (~) following *ACCESS when the file is unlocked. For example, suppose we have a file called HUSH which we want to keep in directory X. To lock this file, we use *ACCESS "X.HUSH" L. When we use *CAT to list the contents of the disk, we will find an entry for HUSH, but not the directory letter, nor the L. To unlock this file, we must type: *ACCESS "~X.HUSH". This means that we have to keep a note of the directory letter for this file before we can unlock it. It's not an infallible piece of security, but it's useful. A more recent version of the AMCOM chip does, however, show the 'L' for locked file. In all of the systems, incidentally, ambiguous filenames can be used.

Renaming files

Occasionally we want to give a new name to a file. We could, of course, load the file, save it under another name, and then delete the old filename. This is not necessary because all that has to be changed is the catalogue entry on the disk. This can be done using the *RENAME command. Renaming is particularly important for data files. Suppose, for example, you have a program which creates an index of names and numbers, and which saves its index to disk under the filename INDX. Now if you place the index program at the start of the disk, and then use it to create a data file, the disk will contain two files, your program and its data. You will still have plenty of space on the disk, so you may feel that you can save a few more programs on it. This will cause the data for your index program, filename INDX, to be sandwiched between two programs. Now if you use the index program again, and create this time a shorter file of data, then it will be saved on the disk using part of the space that was formerly occupied by the old, first file called INDX. If, however, there is not enough space to fit the new data into the old place, you will get an error message 'Can't extend'. This is because INDX has a catalogue position that points to a place with insufficient room. The DFS will not create another file of the same name; it can't fit the new data in, so it comes back with an error message. It's all very reasonable, but annoying.

There are two ways out. One is to delete the INDX file and, if necessary, compact the disk. The other is to rename the offending file. This second approach is rather better, because it preserves the old file, and the new one now finds another place on the disk. *RENAME has to be followed by the old filename, a space, then the new filename, in that order. A Drive number can be included in the first filename (for example, *RENAME :1.B.PROGRAM ECHO). If you put a drive number into the second filename, it will be ignored. Renaming will not occur if the disk is write-protected. As the files are renamed, each original name is printed on the screen with a message to show what it is being renamed to.

The Watford DFS allows you to use ambiguous filenames, so that groups of files can be renamed. For example, if you have files TEXT1, TEXT2, TEXT3, etc., then *RENAME TEXT* OLD-TEXT* will allocate the names to OLDTEXT1, OLDTEXT2, OLDTEXT3, and so on. In the AMCOM system, ambiguous filenames are not used, but *RENAME can be used to unlock a file. For example, *RENAME "~X.HUSH" "A.OPEN" will rename the locked file HUSH as the unlocked file OPEN. The directory letter of the locked file must be known.

Chapter Four
Machine Code and Other Bytes

*SAVE and *LOAD

The ordinary SAVE and LOAD commands of the DFS apply to BASIC programs, and they will, respectively, save from or load to a starting address that is given by PAGE. If you only ever make use of BASIC, and have no application for machine code programs, these commands will be just about all that you'll need in the way of loading and saving. If, however, like the majority of more advanced programmers, you make considerable use of machine code, or if you want to back up machine code programs that you have bought, then some information on what facilities the DFS makes available to you should be useful. We'll start with *SAVE and *LOAD.

The big difference between machine code programs and BASIC programs is that a machine code program may be placed anywhere in the RAM. The normal BASIC starting address of &ØEØØ (cassette machine), or &19ØØ (Acorn or Watford DFS) or &15ØØ (PACE/AMCOM system) can be replaced by any other valid address in the RAM. We can even save, on disk, machine code that is stored in the ROM of the computer, though we cannot load to these addresses. As well as machine code, we can use *LOAD and *SAVE for any collection of bytes anywhere in the RAM. This includes the screen display, and even BASIC programs if need be. We have, in fact, already used *LOAD and *SAVE to shift the position of a BASIC program in the memory (see loading long programs, Chapter 2).

Let's start by considering what has to be done in order to save a machine code program. Now when we do this, we need much more information about addresses than we ever have to worry about when we program in BASIC. Fortunately, all this information is readily available to us when we load a machine code program in from a cassette (or disk). It would also be known, of course, if you had written the machine code program for yourself. What we need to know is the starting address for the machine code program, its length, and what address is its 'execution address'. This last point needs some explanation if you have only programmed in BASIC. Suppose you had a BASIC program which started with lines of DATA, and did not contain any instructions until line 2ØØ. The action of the machine, when it meets a program of this type, is to skip over the DATA lines, and start at the line with the first instruction.

Now, it's possible to create machine code programs which follow a similar pattern. You can arrange for the first few (or many) bytes of the program to be bytes of data which are not to be used until later. In this way the actual program instructions might, for example, start at the twenty-fourth byte, and the address of this byte will be the execution address. Machine code, however, uses no line numbers, and it consists of numbers that are direct instructions to the microprocessor of the computer. This means that all the normal operations of the machine are short-circuited (apart from keyboard and screen). If the instructions to the machine start with the twenty-fourth byte, then you need some way of ensuring that the machine executes the instructions starting at this byte. If this is not done, and the machine starts with the first byte, it will treat this first byte (which is a data byte) as if it were an instruction. There's nothing to stop it from doing this – except to record with the program an 'execution address', which will be the address of the first *real* byte of program as distinct from the first byte that is stored.

To get this information from a program on cassette, you have to type *OPT1,2. This has nothing to do with the DFS, it's part of the normal operating system of the BBC machine. The result of *OPT1,2 is that you get a message at the end of a machine code load which, in addition to the usual filename, number of tape blocks and length, indicates the starting address and execution address of the machine code program. Each of these addresses is given in 'two-word' form, meaning that there are eight hex digits in place of the usual four. If you are using a straightforward disk machine with no 'tube' extensions, you can forget about the first four digits of each address, which will be FFFF. Figure 4.1 shows a typical tape message of this type from a short machine code program. Note that the numbers are in hex with no & mark.

Now, when you have loaded a machine code program from tape using this option, you will have to make a note of the numbers that you see, particularly the starting address, execution address and length. The reason is that you need all of these in order to *SAVE the

```
INDEX 12 1252 FFFFØEØØ FFFF8Ø1F
```

The length is &1252 bytes (469Ø), the first byte is at address &ØEØØ, and execution starts at &8Ø1F. This last number implies that the program is in BASIC.

Fig. 4.1. The detailed information on a cassette program which is revealed by using *OPT1,2.

program on to disk. Type *DISK, and RETURN. Now type *SAVE "filename" SSSS FFFF XXXX, where SSSS is the starting address of the program, FFFF is its finishing address, and XXXX is its execution address. The filename should consist of a maximum of seven characters. Of the numbers, you have noted the starting address, and you need only the four lower digits. The finishing address is the start address plus the length, and the designers of the BBC machine have been exceptionally kind to you in this respect. In place of the actual finishing address, you can enter +LLLL, where LLLL is the length of the program. The + sign is *very* important when you specify a finishing address in this way. The execution address XXXX is, once again, obtained from your notes, and only the lower four digits are used. Many machine code programs use the starting address as the execution address and, in this case, you don't need to use a separate execution address. This makes your command look like *SAVE"filename" SSSS +LLLL, which is a lot simpler. Remember that you must not use the & sign preceding these hex numbers.

This will save the program on the current disk drive (you can specify a drive in the title, along with a directory letter if you choose). If the disk is write-protected, you will get a 'Disk read only' message. If you have too many programs on the disk, you can get a 'Catalogue full' message, and if there is not enough room on the disk, you will get a 'Disk full' error message. These are just what you would expect of any save command, so there's nothing special about *SAVE. You can, however, extend the command by adding another address. This is the 'relocation' address. Relocation means that when the saved program is loaded into the machine again, it will *not* load to the same starting address as we saved it from. Unless you are a dedicated machine code programmer, this will probably seem like a useless type of command, but it has its uses. One is, for example, that it's possible to save the bytes of a screen display (some remarkable chunk of graphics, perhaps), but with a relocation address in the RAM. This means that when the data is loaded again, it does not alter the appearance of the screen, but the data is in the machine, waiting to be used, and you can then reproduce the spectacular screen display at any time. Another application is that you may have created the machine code program while you were using another machine code program (such as a monitor). You may want your own program to occupy the addresses which, at the time of creating the program, are occupied. By specifying a relocation address, you can ensure that the program will relocate when you load it again. Don't use this facility unless you know what you are doing, because not all machine code programs are relocatable! If you use a relocation address, you must also use an execution address.

When a program has been saved using *SAVE, it can be reloaded by using *LOAD. *LOAD has a much simpler syntax. For most purposes, the *LOAD command need only be followed by the filename. Quotes around the filename are optional, but if they are not used, there should be a space between the 'D' of *LOAD and the first character of the filename. You can, for example, type *LOAD"MCPROG" or *LOAD MCPROG, but *not* *LOADMCPROG. If only a filename (which can be a full filename with drive number and directory letter) is supplied, the machine code program will be loaded at whatever starting or relocating address was specified at the time of the *LOAD operation. You can, however, force a program to be loaded to another address. If you use *LOAD NAME XXXX, where XXXX is an address in hex, the machine code (or other) file will be loaded into successive addresses starting with XXXX and ascending in address number. A useful feature of this is that it can be used to verify that there has been no corruption on the disk. By using *LOAD NAME 8ØØØ, the program will be 'loaded into' the ROM addresses. This will not be a true load in the sense that the ROM is not altered, but the internal checking that takes place when a *LOAD is carried out will operate. Because of this, you will get a message which will reassure you that the program is correctly stored, or (very unlikely) that there is corruption. Another possibility, which can be useful for indicating the length of files, is MODE 1, followed by *LOAD NAME 44ØØ. This loads the file to the Mode 1 screen, and places a small pattern on the screen for each byte. The Watford DFS allows an ambiguous filename to be used with *LOAD. If this is done, the operating system examines the files in the order in which they are stored in the catalogue, and loads the first file which matches the ambiguous specification.

Byte saving

The *SAVE and *LOAD commands can be used to save or load any group of bytes that are stored as a single group in the memory. These bytes do not necessarily have to be the bytes of a machine code program; they can be ASCII characters of text, number data, or even the bytes that form the screen display. Any such group of bytes can be saved to disk using the same *SAVE command as is used for machine code. The main difference is that an execution address has no meaning in such a case. If a group of bytes is being saved on disk with a relocation address, so that it will be loaded into a different address, an execution address *must* be supplied. This can be the same as the starting address. It is not used, because there is no *RUN command, but it must be present to satisfy the syntax of *SAVE when a relocation address is being used.

Saving a screen display is best done as part of the program which creates the display. This is because it's simpler to disable the screen messages in the course of a program. It's not particularly desirable, when you play back your dazzling graphics display, to see the disk saving messages appear on the screen as well! These can be disabled by using *OPT1,∅ before the screen pattern is saved, and re-enabled by using *OPT1,1 afterwards. Since the screen data occupies standard address numbers, depending on the screen mode that is in use, you don't have to enquire too closely into the numbers that you use with *SAVE. If you are more adventurous, however, you can try saving only part of a screen, using a smaller range of numbers. This can be very useful if you want to have the same text, for example, displayed at the bottom of the screen while different pictures are displayed on the top half. Another possibility is flicking from one picture to another in a sequence as rapid as the *LOAD process permits. Figure 4.2 shows the standard address numbers for screens in different modes. A useful subroutine for saving and loading screen patterns was given in *BEEBUG*, Volume 1, Issue 6 (page 7). This was for saving to and loading from cassette, but no changes are needed when the DFS is in use. Remember that anything on the disk can be recovered by using a *SAVE command.

Mode	Screen addresses
∅	&3∅∅∅ to &7FFF
1	&3∅∅∅ to &7FFF
2	&3∅∅∅ to &7FFF
3	&4∅∅∅ to &7E7F
4	&58∅∅ to &7FFF
5	&58∅∅ to &7FFF
6	&6∅∅∅ to &7FEF
7	&7C∅∅ to &7FE8

Fig. 4.2. The memory addresses that are used for screen displays. By saving these as files, you can load a screen display at any time in a program.

Using *RUN

The command *RUN, which like several of the DFS commands is also available on the cassette filing system, will cause a named machine code file to be loaded and run. Obviously, this can be used only with genuine machine code programs, and any attempt to use *RUN with bytes of data or with BASIC programs (unless an unorthodox SAVE has been used) will cause the operating system to crash. The facility is useful, since we normally load machine code programs with the intention of running them, rather than listing or amending them. A machine code program cannot, of course, be listed by the BASIC LIST command in any case. What makes the command even more useful is the way in which it can be abbreviated. In the Acorn DFS manual, the abbreviation for *RUN is given as *R. In the Watford system it is shown as *R/. All of the systems including AMCOM, however, allow a much more useful abbreviation, * followed directly (no spaces) by the filename. This makes machine code programs on disk as accessible as any of the operating system commands that can be controlled by the asterisk. For example, you can have *SELRENUM used to call up a selective renumbering program (provided you have a program of that name on the disk!). This requires, however, use of the 'library' portion of the catalogue.

Library, !BOOT and *BUILD

The next set of topics follows from the provision of the *NAME feature of the BBC DFS, the ability to load and run a file that is a machine code program simply by typing its name prefixed by an asterisk. As you might expect, there are some strings attached to this. One of them is that the machine code programs that you want to use in this way should be stored in a special directory, called the library. This is purely a matter of convenience. When the disk system is switched on, the library consists of all files whose directory character is $. Unless you do something about it, of course, all of the files that you put on a disk will have this directory letter. This is a good reason for choosing another letter for your own files. The way the $ directory letter is used is that all files in this directory will be checked when a command such as *Name is typed. If there is a file of this name, it will be loaded and run – provided that it is a machine code program, or that it is arranged so that it can be loaded and run like a machine code program. You can therefore use this directory character for a library of machine code programs from which the machine will make a selection.

The 'library' character need not be $, however. Just as you can change the directory letter, you can change the library characters. The change will, of course, affect only the files that you save from then on. The command that is needed to carry this out is *LIB. The *LIB command can be followed by a drive number and directory letter. If you have only one drive, this isn't so useful, but for two-drive owners, it offers the useful possibility of keeping all the machine code utilities that you might want on one disk, in one drive. Suppose, for example, that you choose to keep all your disk utilities on a single disk, using the directory letter U. Now suppose you put this disk in Drive 1. By typing *LIB :1.U (then RETURN), you will force the machine to search the utility disk each time you use a command such as *FIXIT, which is not part of the normal DFS or operating system set. This will happen no matter which drive you are using currently. Your current drive, which might be Ø, will be restored, along with the current directory letter, whenever the utility has been loaded. Once again, this is a feature which does no more than you could do with a lot of *DRIVE, *LOAD and *RUN commands, but it can save quite a lot of time if you use a lot of utilities. In addition, parameters can be passed to the utility. This means, for example, that if you call a selective renumber utility called SELREN, you could use a syntax such as *SELREN 1ØØØ

2ØØØØ 5. This would mean that you wanted to load and run SELRUN, and carry out the renumbering from line 1ØØØ (old number), making this line renumbered to 2ØØØ, with increments of 5. This is possible only if the machine code utility permits this type of action. The details are of use to you only if you are well acquainted with machine code.

Booting up

There is another useful facility which is also aimed to save time for you. This one concerns the action of the BREAK or SHIFT-BREAK keys. If we consider the normal case, when none of the keyboard links has been closed, the boot action occurs when SHIFT-BREAK is pressed. 'Boot', in this sense, means loading in a file from a disk in Drive Ø. The phrase is used throughout computing to mean some form of start-up action. On some computers, booting means reading in the BASIC language from a disk in Drive Ø. The BBC machine uses BOOT in a wider and more useful sense. When the SHIFT and BREAK keys are pressed, the disk in Drive Ø will spin, and a file whose filename is !BOOT will be loaded. The curious name is an effort to ensure that the file has a unique name that is not likely to be duplicated.

All of this action makes several assumptions. One is that there is a disk in Drive Ø. If there isn't, you may find that the machine locks up, and you will have to press BREAK (by itself) to recover control. Even if there is a disk in Drive Ø, you may find that the SHIFT-BREAK action does not take place. There can be several reasons for this. One is that the way in which you press the keys is important. The designers have done as much as possible to avoid this action occurring by accident. It's not difficult to see why. If you were able to load the !BOOT file by a small slip of the typing fingers, you could find that you had replaced the program or text that you were working on. This could be catastrophic, so the action has deliberately been made rather difficult. You have to hold down the SHIFT key, and jab the BREAK key quite sharply. Pressing both keys down together will not (on my machine, anyway) cause !BOOT to load, nor will a slow press of BREAK with SHIFT pressed. The BREAK key has to be jabbed – a quick press and release – with SHIFT held down. Only this type of action will be rewarded by you hearing the disk start to spin.

That in itself does not ensure that anything more happens. For one thing, there may not be a file whose name is !BOOT on the disk, in which case it can't be loaded. The response to this will be simply to return to BASIC, with the normal > prompt showing. Another possibility is that the file exists, but a code on the disk commands the computer to ignore it! The reaction in this case is the same, the disk stops spinning, and the normal prompt appears.

Does the BOOT fit?

Why should a file called !BOOT be useful? A lot depends on what type of programming you do. For example, you may want each of the programmable keys to carry out some function while you design a BASIC program. The !BOOT file could be a program which initialises these keys the way that you want them. This can be done either by incorporating the key commands (*KEYØPRINT" M for example) as part of the !BOOT program, or by making !BOOT a one-liner which CHAINs another, longer, program for initialising keys. You have a lot of choice in this matter, and it's only by experience that you will find what use of !BOOT suits you best. I use !BOOT as a way of setting some programmable keys and displaying the disk catalogue. There *are* limitations, and we'll look at them later, but for most purposes, it's very useful to get a complete set of actions simply by pressing keys. As a further refinement, link 5 on the keyboard provides for some variation in this action. With link 5 open, as it normally is, the BOOT action occurs only when SHIFT, then BREAK, is pressed as I have described. With link 5 made (closed), however, the BOOT action will take place whenever the machine is switched on and when the BREAK key is pressed (*not* with SHIFT). This, once again, can be useful – but only if your programmer methods suit it. It certainly doesn't suit every programmer to have a file read every time the BREAK key is pressed. The PACE/AMCOM and the Watford systems behave in the same way as the Acorn DFS in respect of this action, but the AMCOM system offers one small but useful extension. When a !BOOT file is selected by SHIFT-BREAK, the AMCOM DFS will automatically set to the correct system. In other words, if the disk has been recorded using Acorn system (SYSØ), then your computer will behave as if you had typed *SYSØ. If the disk is in AMCOM extended mode (SYS1), then your machine is set to this system.

Building a BOOT file

A BOOT file could quite easily be created in the way that you create any other program, then SAVEd with the filename !BOOT. As it happens, however, the DFS permits another way of creating this, or other files, in a much more direct way. This makes use of another new command, *BUILD. *BUILD allows you to make up a program of instructions, most of which will be instructions preceded by an asterisk, and then save the whole file, without going through the usual steps. It's rather like generating a program which uses the disk as a memory in place of the memory of the computer.

An example will make this very much clearer. Suppose that I want to make a !BOOT file that will set two programmable keys for use with the WORDWISE word processor, and which will also give me a disk catalogue. The steps in this process are as follows:

1. Type *BUILD !BOOT.
   The filename must follow the *BUILD command, and the disk that you want to use must be in the current drive. The reason is that when you press RETURN, this filename will be put on to the disk, and space will be prepared for the following file. When this has been done, the number 1. will appear on the screen. This is a 'line number', rather as you would use in BASIC. It is *not*, however, a BASIC line number, and the number is not recorded on the disk. For this reason, you don't need to use RUN to make the !BOOT file execute, and you can't alter one line of the file by itself (not with the normal editing system, anyhow). If you want !BOOT to be a BASIC program, then you will have to type line numbers in addition to the numbers that appear automatically.

2. Type an instruction for the first line of the file.
   In my case, I used *KEYØ !!oc27,45,1 !". This is the instruction which will allow key fØ to start an underline on WORDWISE text, by pressing SHIFT, CTRL and fØ simultaneously. At the end of this line, press RETURN. There is *no* response from the disk, but the number 2. appears under the 1., and you can type another line. My second line was *KEY1 !!oc27,45,Ø !" which has the effect of switching off the underlining. Once again, pressing RETURN causes another line number, 3., to appear. My third line was *CAT. Having pressed RETURN again, to get line 4. appearing, I then pressed ESCAPE to terminate the *BUILD procedure. This causes the disk drive to start, so that the file can

be placed on the disk at the correct address to be located by the name of !BOOT.

It's not over yet. You have to place another instruction on the disk, the one that will cause the correct action to be carried out on the !BOOT file. If you don't do something of this sort, the file will be ignored, because the computer needs to be instructed whether this is to be loaded (as a machine code program), *RUN (only possible if it is a machine code program) or *EXECuted (something which is useful for BASIC). Of these choices, we have dealt with *LOAD and *RUN already, but unless your experience of BBC BASIC is extensive, you may not have met *EXEC. This is a command which loads a file just as if the commands of the file had been typed at the keyboard. In other words, the action will be to load and run, just as if you had typed each line of the file as a set of direct commands, and pressed RETURN at the end of each line. The options are set by the *OPT 4 command:

```
*OPT 4,Ø will cause the !BOOT file to be ignored.
*OPT 4,1 will *LOAD the !BOOT file.
*OPT 4,2 will *RUN the !BOOT file.
*OPT 4,3 will *EXEC the !BOOT file.
```

Any of these commands will, when you press RETURN, cause the disk to spin so that the code for the command can be recorded on the disk. This option is shown in the catalogue display when you use *CAT. Now when SHIFT-BREAK is pressed, this code will be checked, and the appropriate action carried out. In the example, since the !BOOT file consisted of a set of ordinary commands (*not* machine code), the *EXEC option is the one I need. Typing *OPT4,3, then RETURN, places this on the disk. The disk, of course, must not be a write-protected one.

As a result, when I press SHIFT-BREAK, the disk in Drive Ø spins, the programmable keys are set up for me, and the disk catalogue appears. This is a rather timid and brief example of !BOOT, but you can go very much further. You can, for example, end a !BOOT file with a line such as CHAIN NEWPROG, so that a program which you are working on is loaded and run. The !BOOT file will, before this happens, have set your programmable keys and carried out any other actions that may be needed. You can even call up alternative operating systems, such as WORDWISE (using *W.) as part of a !BOOT file, but some care is needed. If you want to call up a system such as WORDWISE which exists in a ROM, then you have to make this the last instruction of the !BOOT set. You cannot have lines such as:

```
5. *W.
6. *CAT
```

because the action of calling up WORDWISE destroys the rest of the file. You can, however, have:

```
5. *CAT
6. FOR N=1 TO 1ØØØØ:NEXT
7. *W.
```

which will display the catalogue for the disk, and then wait for long enough to allow you to read the catalogue. The program will then continue, and the WORDWISE menu will appear. You might think, incidentally, that you could use a line such as:

```
6. REPEAT:X=GET:UNTIL X=32
```

to make the continuation depend on pressing the spacebar. Alas, it seems to have odd effects on the operating system. On my machine, it caused a spurious GOTO6Ø to appear, followed by a 'No such line' message. Substituting *WORDWISE for *W. in line 7 causes a different error message, but the scheme was still unworkable, presumably because the keyboard buffer is being used in the EXEC function.

The PACE/AMCOM and Watford systems treat the *BUILD command in the same way as the Acorn DFS, though the PACE/AMCOM system permits the use of longer filenames for a BASIC file that is loaded by !BOOT.

*DUMP, *LIST and *TYPE

Three other commands can be dealt with at this point. They apply to any file that is stored on the disk, whether it is !BOOT, any other file created by *BUILD, or anything else. As we'll see, though, they are particularly useful for files that consist of only text characters. We'll start, alphabetically, with *DUMP. This has to be followed by filename, and the action is to print on the screen the listing of the file. This is given as hex numbers on the left-hand side of the screen, and as characters on the right-hand side of the screen. Only hex numbers between &2∅ and &7F produce characters on the screen, and any number outside that range is shown on the right-hand side of the display as a dot. This is the best way of finding out what a file consists of. If there is a character for each number, with dots only where a full-stop occurs (ASCII hex code &2E), then the file is purely a text file, not a BASIC program saved by SAVE, or a machine code program. The distinction about the BASIC program arises because BASIC can be saved in two ways. A BASIC program that is saved by SAVE uses codes, called 'tokens', in place of keywords like PRINT and TAB, etc. Each of these 'tokens' is a number in the hex range &8∅ to &FF, outside the ordinary ASCII range. A BASIC program can, however, be saved as a set of ASCII codes by using the *SPOOL command, in which case, each character will be an ASCII character, with no tokens. A BASIC program that has been saved in this form cannot be loaded for running by the LOAD command, only by the *EXEC command. That's why we had to use the *EXEC option for the !BOOT file in the example, because *BUILD makes only text files – it does not cause conversion into BASIC token form.

*DUMP is therefore a useful way of finding out what sort of file you have. Once you know what you are doing, you can carry out useful editing actions on such a file. For the moment, though, we'll concentrate on the facilities that are available. If you find that the file is a text file, there are two other commands which you can use. One is *LIST. *LIST has to be followed, as you might expect, by a filename. When you press RETURN, the file will be listed with each line numbered (the sequence is 1,2,3...) just as it is when you create a file using *BUILD. This is a particularly useful way of checking what is included in your !BOOT program. The alternative is to use *TYPE, which gives the same results as *LIST *but* without the line numbers. You should not use these commands on a file that contains characters that are out of the normal range of &2∅ to &7F. The reason is that characters whose codes range from &∅∅ to &1F are control codes, which can carry out actions such as clearing the screen, shifting text windows, and so on. If a collection of characters of this sort reaches the screen (more correctly, the VDU drivers), then the results are, to say the least, unpredictable. That's why it's desirable to check what type of file you have before using *TYPE.

The combination of *BUILD and *TYPE can be used as a crude type of word processor. You should not, however, attempt to read WORDWISE files using *TYPE unless you are sure that the files do not contain any embedded commands. A WORDWISE file that has been saved by its SPOOL option is safe to read by the *TYPE command, but not one which has been saved in the normal way. The AMCOM DFS does *not* send non-ASCII characters to the VDU, so that the use of *LIST and *TYPE is less of a hazard with this system.

Chapter Five

Text Files and Their Problems

Text files

A text file means a set of ASCII codes stored on the disk or in the memory of the computer. The *BUILD command is one which we can use to create a text file. This command numbers the lines of the file, but these numbers are not recorded – they exist purely on the screen as a convenient way of finding your way round the file. In addition, each letter of the text is not recorded at the time of typing, or even when you press RETURN. This would be a very wasteful procedure, because it would mean spinning the disk and finding a vacant place simply to record one ASCII code. Instead, the computer places the codes in a piece of memory that is called a *buffer*. A buffer is a piece of memory that is allocated for temporary storage. In this case, the capacity of the buffer is 256 bytes. This is the size of a sector, and the contents of a buffer will be recorded on to the disk when the buffer is full, or when the ESCAPE key is used to terminate the *BUILD process. Text files of this type can be read, as we have seen, by making use of the *DUMP, *LIST or *TYPE commands.

The *BUILD command, and its associated text reading commands, can be used for elementary word processing, but a much more satisfactory solution is the use of a true word processing program. The Acorn word processor, VIEW, is available at the time of writing, but many users of the disk machine are likely to have had experience of WORDWISE, the word processor which has been available for a longer time. The description of word processing in this chapter, therefore, refers mainly to WORDWISE, but because text files are much the same no matter how they were created, the advice that is given here will apply to VIEW and to other systems. It is even possible, as we shall see, to read text files that have been created with other word processor programs on other makes of machines!

WORDWISE and the DFS

WORDWISE, in case there is any reader who does not know of this program, is a word processing program in ROM form. The ROM chip fits into one of the vacant sockets in the BBC machine, and it is switched into use by typing *W. (or the full version, *WORDWISE). You must be careful not to have files in the library which are called W.! The WORDWISE chip was on the market before the disk systems for the BBC machine were widely available, but the program is fully disk-compatible.

The commands of WORDWISE are not changed in any way when a disk system is in use. There are, however, some quirks that are caused by the substitution of the WORDWISE chip for the normal machine routines. One is that filenames in lower-case are taken as being different from filenames in upper-case. Whereas the ordinary DFS use will treat the file NAME as being the same as the filename name, WORDWISE does not. This means that if you save a file called NEWBOOK, it can't be overwritten by one called newbook. This is not true of BASIC programs – a file called myprog will replace one called MYPROG. Another point is that you can add the directory and drive numbers to a WORDWISE filename just as you would normally. When WORDWISE saves, and prompts for a filename, you can type, for example, :1.T.MYTEXT. The result of this will be to save the text on Drive 1, using directory T, with the filename MYTEXT. You can also use, for example, T.MYTEXT to save to the current drive (or your only drive) with directory letter T. As always, if you use only the filename, the text will be saved on the current drive, using the $ directory. One oddity that I have noted when using WORDWISE with the current version of the AMCOM DFS is that there is no 'Disk full' error message – just a 'faulty drive' message when the disk is full.

WORDWISE and BASIC

As we noted in Chapter 4, a BASIC program saved in the usual way is *not* a text file. This is because command words like PRINT, TAB, INPUT and so on are not stored as a set of ASCII codes, but as non-ASCII tokens. This makes a BASIC program occupy a much smaller space in the memory than we might expect, but it means that it's difficult to convert back into readable form. As it happens, however, the BBC machine allows us to store a BASIC program on disk (or on cassette) in purely ASCII form. This is done by using the *SPOOL command. When the first reports of the use of this command were printed, they were rather garbled, so many readers may not be aware of the correct syntax. The following instructions set out in detail how a BASIC program is stored in ASCII form. The command that is needed is *SPOOL.

1. Check the program by listing it, and have a disk ready in the currently selected drive.
2. Type *SPOOL FILENAM, where FILENAM means the filename that you want to use for the program. Remember the 7-character limitation with Acorn and Watford systems.
3. Type LIST. This will cause the program to list on the screen, and the ASCII codes will be saved to the disk at intervals. The saving takes place each time the buffer is full, so that one disk sector is being written each time this happens.
4. Type *SPOOL. It is *very important* not to type a filename this time. This second *SPOOL has the effect of saving whatever text is left in the buffer, because it would be most unlikely that the last piece of text would fill the buffer. If you use a filename for the second *SPOOL, you will empty the buffer, and start again!

BASIC that has been saved in this way is in text form, and it can be read by *TEXT or, more important, by WORDWISE. The features of a word processor can be used for editing in ways that the standard editing methods of the BBC machine cannot match. Many users, in fact, create and edit all of their BASIC programs on WORDWISE simply because the editing features of WORDWISE are so much simpler and more powerful than those of BASIC. You don't, for example, have to copy a whole line just to eliminate one incorrect character. The 'search-and-replace' editing action of WORDWISE also allows you to make several changes by means of one command, something that is not possible with the ordinary editor.

When a BASIC program has been edited with WORDWISE, it can be saved in the usual WORDWISE way, as a text file. To load it back into the machine as a BASIC program, you have to use *EXEC, followed by the filename. This will load in the text, convert command words into tokens, and store the BASIC program into the memory in its abbreviated form, ready to run. When *EXEC is used, the memory is not cleared for loading, as it would be by LOAD. Because of this, it's possible to merge two programs. If you have a program already in the memory, and you use *EXEC NEWPROG, then the lines of NEWPROG will be added to the existing program lines. If the first program used line numbers 1∅ to 1∅∅∅, and NEWPROG uses line numbers 2∅∅∅ to 6∅∅∅, then the programs merge into one. If there are two lines with the same number, however, only the one that was loaded with *EXEC will remain in the memory.

WORDWISE itself has a SPOOL-saving option. This is not for use with BASIC programs, but for storing WORDWISE files as purely ASCII files. In normal use, formatting commands are added to text in a WORDWISE file. We use ti2, for example, to mean a temporary indent of two spaces. This command is used in place of the actual two spaces. Other commands like ce2 (centre two lines) take up very much less space in the memory than the spaces would. These formatting commands do not use ASCII codes, otherwise the words would be printed instead of controlling the print action. When the WORDWISE text is stored, then, there will be a number of bytes of code which do not correspond to ASCII coded characters. If you want to record a file that is entirely ASCII, you have to use the SPOOL option of WORDWISE (option 8 of the WORDWISE menu). The effect of this will be to carry out all the formatting commands, adding a large number of spaces (ASCII &2∅) to the text. A file of this type contains no formatting codes, and so it can be read by means of *TYPE, or by a BBC machine which does not have the WORDWISE chip. A further advantage of using this option is that it allows text to be fetched from disk automatically. For example, suppose that we have saved some text, using the SPOOL option, under the filename of "MIX". We can then create a new file and, at some place in it, type the command GF"MIX", using the normal WORDWISE embedded command method. When the main file is printed, the GF"MIX" instruction will cause the disk to load in the MIX file, print it, and then return to the main file. Note, however, that this is possible *only* if there are no embedded commands in the MIX file. If this is not a SPOOLed file, then the presence of the embedded commands will cause odd effects – some commands will be printed and others may cause the printer to hang up. Another use for this option is to save a large file on disk in its formatted form, and simply use GF to call it up, so that the printer and the disk are working together. This can be one method of combining very long files. If you have stored several long files using names such as FILE1, FILE2, FILE3, then a WORDWISE 'text' which simply consisted of

```
GF"FILE1"
GF"FILE2"
GF"FILE3"
```

would make one continuous printing of the files from disk to printer. Remember, however, that once a file has been stored in SPOOL form, it cannot easily be restored to the ordinary form with embedded commands. This will make it more difficult to work on, if you feel that it needs further editing.

The memory of the BBC machine with a DFS installed, permits about four thousand words at most to be stored internally as WORDWISE text. This is about right for a short chapter, but for long chapters, or for a pamphlet which is not subdivided into short chapters, it is inadequate. Another way round it is to save a long chapter as two sections. This gets around the problem of allowing further editing. By using the preview (Menu option 7), you can find out how many pages the words will occupy. Print to the nearest number of whole pages, then delete the text that has been printed. Shift the cursor to the end of the text, and use Menu option 4 to load the next part of the text to the cursor position onwards. Then put in the new page number, and resume printing.

Cassette problems

Because WORDWISE was ready for marketing before the BBC machine was in a suitable condition to use it (WORDWISE, like the disk system, needs the 1.∅ or higher operating system), many users of WORDWISE will have text files on cassette which have to be transferred to disk. The problem here is that many of these cassettes may contain book chapters, articles or other documents which came close to running out of memory on the cassette machine. These will not load into the disk-equipped machine for transfer to disk. We have already looked at methods for loading in long BASIC programs, but the problem of long BASIC files is a rather different one. For one thing, you can't simply leave WORDWISE, reset the value of PAGE, and then go back to WORDWISE. Each time you return to WORDWISE, the value of PAGE will be reset to the value that is used by the disk system. The problem is not so acute for users of the PACE/AMCOM DFS, because the value of PAGE for this system is &15ØØ, resulting in not so much memory being taken up. Nevertheless, it's highly likely that if you have a lot of cassettes of WORDWISE text, you will have some that are too long to load into a disk-based machine. Fortunately, it's possible to recover all of the text from even the longest WORDWISE cassette files, but the method requires a bit of explanation, and some patience!

The principle is simple enough. It involves switching to BASIC (not WORDWISE), setting the PAGE number to the address (&ØEØØ) that is used by the cassette system, and loading in the WORDWISE file from cassette just as if it were a machine code program. This involves the use of *LOADØEØØ to force the bytes of text to load into consecutive addresses starting at the PAGE address of &ØEØØ. Remember that the *LOAD instruction does not require the & sign with hex numbers. When the text has been loaded in, it is then saved back on to *another* cassette in two halves. This has to be done on to a cassette, because with PAGE set to &ØEØØ, we cannot make use of the DFS. These separate halves of text files are then read by WORDWISE, having switched back to disk operation, and can be saved as two separate files on disk. Let's look at the steps in detail.

1. If you are using WORDWISE, save any text and return to BASIC. Type *TAPE (RETURN), then PAGE=&ØEØØ. This restores the conditions that exist when only the cassette system is present. If you use BREAK, however, the disk system will be restored, so keep your finger away from that BREAK key!

2. *LOAD"TEXT" ØEØØ (RETURN). This will load, from a cassette, a file called TEXT. Needless to say, you will use the correct filename for your own file. You can now sit back and wonder how you ever managed to put up with cassettes. When the cassette has finished loading, take a note of the displayed numbers. The second one is the interesting one, because it's the length of the file. Note this number, which is in hex.

3. Add the length number to the starting address of &ØEØØ. You don't need to be good with hex arithmetic, because

   ```
   PRINT~(&ØEØØ+&length)
   ```

   will give you the final address in hex. In addition you need to find a midway address. Do this by using:

   ```
   PRINT ~(&ØEØØ+.5*&length)
   ```

This automatically takes the INT of the number, so that it works just as well when the length number is odd. In each case, 'length' means the length number in hex. Note the end address and the midway address.

4. Put a fresh cassette into the cassette recorder, remembering to wind on the leader (well, you might have forgotten these details by now!). Type *SAVE"FILEA" ØEØØ midaddress, where midaddress is the hex address in the middle of the file. Press RETURN, then press the RECORD and PLAY keys of the cassette recorder. Wait until this half of the text has been recorded.
5. When the recorder is finished, type *SAVE"FILEB"midaddress endaddress. In this case, endaddress is the final hex address. Press RETURN, then the cassette recorder keys, to save this file also on tape.
6. Type *W., select the New Text option of WORDWISE, then type *TAPE (RETURN).
7. Load in FILEA using WORDWISE load option 2. Then return to disk operation by using *DISK. Now use WORDWISE option 1, with a suitable filename, to save this text file on the disk.
8. Type *TAPE again and load in the second file, FILEB, using WORDWISE option 2.
9. Type *DISK, and use WORDWISE option 1 to save this also on disk.

As a result of all this hard work, you will have the file, split into two sections. The character at the end of one section will appear at the start of the next. You cannot be sure that you will not have split a word, but at least you will have saved all of the file on a disk. Some minor editing will then put the files into order. It's tedious, but you have the satisfaction of knowing that it won't have to be repeated.

Reading other systems

Since the BBC machine has been bought by many who, like myself, cut their teeth on a different make of computer, it's possible that some readers may have text disks that were prepared on other machines, such as the APPLE 2, TRS-80 and so on. Some of these disks *can* be converted into BBC text disks, but it's not easy to generalise about this. If the disks are of the same track standard (for example, both 40-track) and the text files on the old disk are purely ASCII (that is, no embedded control codes), then there's a sporting chance that the method I shall describe will allow you to read a substantial amount of the text from the old disk. As you can probably guess, I have tried this with some success. I was using disks which were prepared in the USA using a TRS-80 Model 1 with disk system, and with the Electric Pencil word processor. Now the Electric Pencil, which was at one time the most popular word processor program before WORDSTAR was released, does not use embedded control characters. Instead, the whole of the text is recorded as an ASCII file, and the formatting instructions are added just before printing but form no part of the text. This is ideal for our purposes, because it's not nearly so easy to deal with text that contains non-ASCII characters.

The principle is fairly straightforward, and similar in some ways to the method that I have just described for transferring long files from tape. We load bytes from the 'foreign' disk (the source) into memory. We then save these on to a destination disk, using a new filename. When we have gathered a disk full of such files, we then read them, one by one, into WORDWISE, edit as necessary, and save again, this time as a finished WORDWISE file. It's hard work, but if it allows you to use 40,000 words that you thought you might have to type again, it's effort well spent!

Now the first part of this is the most difficult, because the source disk has been prepared on another machine. A TRS-80 disk, for example, has its catalogue on one of the middle tracks, instead of on the outer track as the BBC DFS does. There's no question, then, of simply typing *LOAD and a filename to get some bytes stored in memory. What is needed is a disk editing utility. This is a machine code (or BASIC) program which will read in bytes from a disk, sector by sector, and place them in order in memory. I'll deal with disk utilities in more detail in Chapter 6. All I want to say at the moment is that it's possible to specify a starting address in memory (I used &1C∅∅), a starting sector (the first one is ∅, not 1), and a number of sectors (I used &4∅). These are typed in, and pressing RETURN, with the source disk in the current drive, will start the program reading sectors. Each byte is loaded into memory, starting at the specified address. The result, using the figures I have quoted, is to fill the memory with bytes from address &1C∅∅ to &5C∅∅.

Once bytes from the disk have been loaded into memory, thanks to the disk utility program, the rest of the task is comparatively plain sailing. Put a new (formatted) disk in the drive (or into a different drive number), and select this disk. Then use *SAVE TEXT1 1C∅∅ 5C∅∅ to save the bytes from memory to the disk. That's one batch. Now this first batch saved the bytes from sectors &∅∅ to $3F. The next *SAVE should specify a starting sector of &4∅, and should load another &4∅ sectors into the same range of memory. This can now be saved on the destination disk as TEXT2, and so on. The last load from the source disk will be from the starting sector of &18∅, and for only &∅A sectors, rather than &4∅. This assumes a conventional 40-track disk. Because the number of bytes that you have copied may be more than you can record on a BBC disk, you may have to spill some material over on to another destination disk. You should end up with a destination disk or disks that will contain the same bytes, but recorded on the BBC machine and with a catalogue in the right place.

You will, of course, have recorded the whole catalogue of the source disk, and it's highly likely that this will contain some non-ASCII codes that your own computer will not take kindly to. The important thing is to ensure that these do not spoil the transfer. The next step, then, is to get back to the disks that you have created – we'll call them the salvage disks now. Use the disk utility program to examine the characters on the first track of a salvage disk. If this contains nothing but ASCII codes, and they make sense (language, not gibberish), then it looks as if this is not a catalogue track. If the catalogue is not on the first track, then it's most likely to be on the middle track, which means that we can read about two hundred sectors from the disk before running into trouble. Use WORDWISE, then, to load TEXT1 from the salvage disk. This, if the old source disk was pure ASCII code, should be text that you can edit and generally slap into shape, with no embedded commands. If it does have embedded commands of its own, then you will have to root them out. One popular pattern is to have an embedded command on a new line following a full-stop. This type is quite simple to remove – if you can find it. The trouble is that the coding may not be visible on a WORDWISE edit, because the commands like .ti2 may have been converted to non-ASCII codes. Looking on the bright side, however, I haven't met one yet that I haven't been able to work with. The only really awkward part of the process occurs when you find the catalogue track, because you will have to try avoiding this portion. You may, if the worst comes to the worst, have to use the utility again to record the sectors up to and also beyond the catalogue sectors, but omitting the catalogue.

Trying to load other types of file from disk is not such a good prospect. Once again, BASIC program saved in ASCII form may be loadable by following the procedure detailed above, saving as an edited WORDWISE file, and then entering into the machine by using *EXEC. The WORDWISE editing will have to be used to change the BASIC of the 'foreign' program into BBC BASIC. In this respect, the 'global' seek and replace commands of WORDWISE are extremely useful, because they allow you to find and change each word of the other BASIC which would cause a syntax error in BBC BASIC.

Chapter Six
Disk Utilities and How to Use Them

The utility disk

A *utility disk*, as the name suggests, is a disk on which are stored several routines, mostly in machine code, which should be of use to disk users. A disk of this type is sometimes advertised under various names. For example, the disk-editor by Computer Concepts is advertised as the *DiscDoctor*. Whatever the title, the aims are always to supplement the facilities that are available in the DFS. The Acorn utility disk, for example, contains the formatting programs which are not present in the ROM. In this chapter, however, we shall be concerned with the more advanced features of a utility disk. These are features which allow us to recover information from a damaged disk, and to read (see Chapter 5) disks that were made on other machines. The most valuable feature, then, for a utility disk is a disk sector reader and editor.

The disk sector editor

The purpose of a disk sector editor is to allow you to see what bytes are stored in any sector of a disk, and to modify these bytes if necessary. Simply viewing the bytes is not harmful, but careless changing of bytes can make a disk almost unreadable, so some caution has to be exercised. If you are likely to want to make use of a disk sector editor, it pays to experiment first with a spare copy of a disk. Never work with a sector editor on any disk for which you have no back-up of any kind, unless you are certain that you will only be reading sectors, not changing them. Some disk editors make it intentionally difficult for you to alter bytes. Others make it all too easy!

To avoid being too general and therefore too vague, I shall describe the facilities that are available on one utility disk – the one that is supplied with the PACE/AMCOM DFS – and comment also on the Watford utility disk. Since the principles of utility disks are broadly similar, a detailed description of one will serve as an outline for all.

AMCOM utility menu

The AMCOM utility disk contains a large number of routines, several of which are not specifically aimed at disk users. These include, for example, programs to dump screen graphics on to EPSON or NEC printers. At this point, we shall concentrate on the disk editor which is loaded (AMCOM system) by *UTILITY rather than on these other programs, which also qualify to be called utilities. The Menu for the UTILITY program is shown in Fig. 6.1. It

```
      Amcom Disc Utility
    Type:
1   to load sectors from disc
2   to save sectors to disc
3   to edit the disc
4   to verify a track
5   to change the current drive
6   for a search through memory
7   to edit memory
```

Fig. 6.1. A typical disk utility program. This is a copy of the menu for the AMCOM utility.

consists of seven items, none of which provides any clue about how to return to BASIC. This is done by pressing the ESCAPE key, and will always be accompanied by a 'Bad program' error message. There is no corruption of memory, however, when this happens, so that procedures which involve switching between the disk editor and BASIC (as detailed in the previous chapter) do not corrupt data. Data will be corrupted, however, if it is loaded into the same memory space as the disk editor. For this reason, it's always wise to type *OPT1,2 before loading a utility for the first time if it can be loaded without running. This allows you to see the length of the file, so that you can calculate the last memory address that will be used by the utility. A utility will normally occupy the lowest part of the

RAM that is available. For the Acorn and Watford systems, this means &19ØØ on. For the PACE/AMCOM system, it means &13ØØ on (which is where the AMCOM utility loads). A few utilities, however, load into the highest available part of memory. The use of *OPT1,2 will give you the information. Write this on the disk cover as soon as possible, because you will usually find that you want it just when you can't possibly go through the motions of typing *OPT1,2 and loading the disk again. If the utility is corrupted, you may lose control of the machine and have to use BREAK, then reload. There is always some risk that this process may corrupt a disk if you have been altering bytes, so care is always needed. If you have a back-up of the disk, it should always be possible to recover from such problems. Remember that the *BACKUP facility of the DFS will carry out an exact sector-for-sector copy of a disk.

With that advice given and digested, let's look at the utility commands in detail, though not necessarily in order. For anyone with more than one drive, option 5 of the AMCOM disk is useful. This allows you to change drive without having to leave the utility program. When this option is selected (press '5' without using RETURN), the screen prompts for a drive number. When this is given and RETURN is pressed, the message 'press any key to return to menu' appears, and pressing a key will get back to the main menu. Another useful feature allows you to check the health of a disk *without* any damage to the information on it. The formatting procedure tests tracks, but will completely destroy any data that is stored on them. Option 4 of the utility allows a given track (you have to enter the track number in hex) to be checked, and the data replaced after verification. A screen message will then report on the health of the track. If the track is faulty, it's possible that the data cannot be replaced, or that it was incorrect in the first place. In such a case, you will have reason to be glad of a back-up disk.

Editing the disk

Different utility programs use different ideas of what is meant by editing a disk. Some programs take this to mean that each byte can be read and a decision made whether to leave it or change it. Another interpretation is that the editor simply prints on the screen the bytes in each sector of the disk, with no provision for change. The PACE/AMCOM utility adopts this second approach. When you select the Disk Edit option (3), you will be prompted for a sector number. This will be the number of the first sector that will be displayed. If you have one drive, you will have to be sure at this point that you have removed the utility disk, and inserted the disk that you want to investigate. If you have more than one drive, you will have to be sure that you have used the 'change drive' option so that you are not editing the utility disk! This is such a nightmare as far as I am concerned that I keep two back-up copies of the utility disk!

Following the prompt, you have to enter the sector number at which you want editing to start. If you are using the editor for the first time, then the most useful sector to start with is ∅. The numbers are in hex, as is usually the case for any machine code program that is concerned with probing disk or computer memory. The point of selecting Sector ∅ is that this allows you to see what is placed in the first part of the catalogue. The action is started by pressing RETURN. The drive spins, and a listing is shown on the screen. This listing is of Drive number, Track number, Sector number, Byte number, hex value and ASCII character (if relevant). Only one byte is shown, and to make the next one appear, you have to press RETURN. This does *not* cause the disk to spin again, because the action of the disk editor is to read the bytes into a buffer memory, one sector at a time. Each time you press RETURN, you will see what is stored at some part of this first sector. If you hold the RETURN key down, you will see the listing scroll. When byte &FF has been displayed, the disk will spin again to load in Sector ∅1, and the bytes of this sector will be displayed. It is possible, if you have a few hours to spare, to go through every byte on the disk in this way. In general, pressing ESCAPE will return you to the menu, and pressing ESCAPE while the menu is displayed will return you to BASIC. There will also be an error message because of the use of addresses below the usual BASIC limits.

The Watford disk editor is entered by typing *EDIT. The result is to display a selection on the screen, with the cursor placed at the words 'OS Command'. This allows you to make use of any operating system command, such as CAT, OPT, etc., by typing the command (you don't need to use *) and then pressing RETURN. If you type BASIC, then RETURN, the machine will return to BASIC. The BREAK key returns you to the menu. Choice of a track and sector to edit is done using the TAB key, which will move the cursor down the sector and track numbers. To read a sector that is selected by the cursor position, you press RETURN. You can also specify a filename, and get the first sector of that file.

When a sector has been chosen and read from the disk, it will be displayed on the screen. Each line of the display shows sixteen bytes of the sector, using both hex and ASCII characters. Using the TAB key will switch the cursor between the hex and the ASCII version of the same byte. The cursor keys now permit you to place the cursor over any byte, and you can change the byte by typing a new digit or pair of digits. The change is *not* made on the disk itself until you press the COPY key. Pressing the ESCAPE key will stop the editing process without causing the results of editing to affect the disk. If you only want to look at what is on a sector, keep your hands off the keys apart from the cursor control and ESCAPE keys!

In general, though, you will be looking through a sector for some specific purpose. This will usually be to check a catalogue entry or to check that data is correct. Since the catalogue is on the first eight sectors of an AMCOM extended system disk (first two sectors only for the Acorn disk), we can look at the characters of a typical catalogue entry, Fig. 6.2. This is for an AMCOM SYS1 disk, and it is a printout of the information for Sector ∅∅, bytes &2∅ to &3F. These sixteen bytes contain the catalogue information for the first file on the disk. The Acorn system would reserve bytes &∅8 to &∅E for the filename of this first file, byte &∅F for its directory letter, and bytes &∅8 to &∅F on Sector ∅1 for the rest of the catalogue information. The Watford DFS will follow the Acorn system, but if you have chosen to use the 62-name catalogue, it will use Sectors ∅3 and ∅4 in Track ∅ for an extra catalogue. The information on the Acorn system, therefore, applies almost exactly to the Watford system's first catalogue, and the second catalogue of the Watford system is organised in the same way as the first, but using two more sectors. The differences are less than they appear – the information is pretty much the same, it's only the way that it is stored that is different.

The catalogue data

The most important feature of the catalogue sectors is the information on where data is stored. Take, for example, the data that is shown in Fig. 6.2. for an AMCOM file. This is the first file on the disk, and its catalogue extends from &2∅ to &3F. Of these 32 bytes, &28 to &36 are used for the filename. This space permits a filename of up to fifteen letters, one of the advantages of this system.

```
20          00

21          13

22          00

23          13

24          00

25          08

26          00

27          08

28          55          U

29          54          T

2A          49          I

2B          4C          L

2C          49          I

2D          54          T

2E          59          Y

2F          20

30          20

= =         = =

= =         = =

37          24          $
```

Fig. 6.2. The appearance of a disk edit display. This shows the first catalogue entry for an AMCOM disk. Bytes up to &2∅ are used in this sytem for information relating to the disk as a whole.

The directory letter is stored in &37. The rest of the data is concerned only with storage.

This part is not so easy to unravel. The reason is that the number of bytes stored on a disk can easily exceed &FFFF, the maximum hex number that can be stored in two bytes. The system must therefore provide for larger numbers by storing more digits somewhere else. Another two (binary) digits, corresponding to hex digits ∅ to 3, are therefore stored at another address. For the first file on a disk, this should not have to worry you, because the length range should be well within the limits of &FFFF. In such a case, the start sector number is shown in byte number &27, the load address in bytes &2∅ and &21, and the execution address in &22 and &23. The length of the file is in bytes &24 and &25. Wherever two bytes are shown, numbers less than &∅1∅∅ will fit into one byte only, and this is *always* the first of the two numbers. Figure 6.3 shows an analysis of a typical entry into this first catalogue.

Byte no.	Value	Meaning
24	38	} &∅238=568 bytes
25	∅2	
. .	. .	. .
27	∅8	starts in sector ∅8
. .	. .	. .
37	24	directory $

Fig. 6.3. Analysis of the most useful parts of a catalogue entry. This is a comparatively small program, as most are.

Byte &26 has a special part to play only if a very long file, or one with a special load address is in use. Byte &26 is divided into four parts. One byte consists of eight binary bits (Fig. 6.4), and address

Composition of &26 (AMCOM) or &∅E (Acorn)

x	x	x	x	x	x	x	x	bits (8 in a byte)
7	6	5	4	3	2	1	∅	place numbers

Note: For a pair of bits:

∅∅ means ∅
∅1 means 1
1∅ means 2
11 means 3

Fig. 6.4. One byte is used to take an 'overflow' from other bytes. This byte uses its constituent bits in pairs so as to add a digit between ∅ and 3 to a hex number of four digits.

&26 uses these in twos. The lowest two (right-hand side), bits ∅ and 1, are used to extend the number of the start sector. Since there are &192 sectors on a 40-track disk, it would be possible to have a short file stored starting at sector $191. The &91 part would be stored in address &27, and ∅1 as the last two bits of address &26. For an 80-track disk, there could be $32∅ sectors. A file which started in sector &31F would require &1F to be stored at byte &27, and the bits 11 as the last two bits of &26. This is because 11 binary is the number three, denary or hex.

A similar system is used for the length of the file. Bytes &24 and &25 hold the (hex) numbers for a four-digit file length. This is satisfactory for files up to &FFFF in length – which is longer than any file that could normally be saved from the memory of the machine. The system has to allow for the extra memory of the 'tube', however, or, more immediately, for utility programs which allow separate files to be joined on the disk. Because of this, the two bits whose place numbers are 4 and 5 in byte &26 are used. These add hex numbers ∅ to 3 ahead of the number stored in &24 and &25. Load and execution addresses on the normal machine will always be four-digit hex numbers, but bits 2,3,6 and 7 of the byte &26 are reserved for higher numbers. These numbers would, once again, be used if the 'tube' were in action. If bits 2 and 3 in &26 are both 1, then the file will be loaded to the second processor at the far end of the 'tube'.

Now all this information is also recorded in the Acorn catalogue, but in different form. The name for the first file is in Sector ∅, bytes &∅8 to &∅E, with the directory letter in &∅F. The information, however, is in the same byte numbers of Sector ∅1. The start sector number is held in byte &∅F, sector 1. If necessary, its extra two bits will be bits ∅ and 2 of byte &∅E. The length of the file is held in &∅C and &∅D, with two more bits which are bits 4 and 5 of &∅E. The load address is held in &∅8 and &∅9 (bits 2 and 3 of &∅E used for 'tube' loading), and the execution address is held in &∅A and &∅B (two more bits in places 6 and 7 of &∅E for the 'tube').

That covers the first file – how about the second? This is where the advantage of working in hex numbers becomes obvious. In the AMCOM system, with the first file information in &2∅ to &3F, the second file information is in &4∅ to &5F, the third in &6∅ to &7F and so on. With the Acorn system, the filenames (plus directory letter) are in &∅8 to &∅F, &1∅ to &17, &18 to &1F, and so on in the first sector. The load, execute, length (etc.) data is held in the same addresses of the next sector (Sector ∅1). The Watford system uses sectors &∅2 and &∅3 as a pair in a similar way to the use of &∅∅ and &∅1. Within each set of catalogue data, the positions of the bytes are exactly the same. For example, in the AMCOM system, the start sector bytes are &27 (first file), &47 (second file), &67 (third file) and so on. In the Acorn system, Sector ∅12 contains length bytes in &∅F, &17, &1F, &27, &2F, etc. Whichever system you happen to be working in, therefore, you should be able to extract the important data. Of this, the length and start sector numbers will be the most important if you want to make alterations to a catalogue entry. Keeping for the moment to the role of a spectator, though, you can note the start sector and length of a file that you are interested in. You can then use the length number to work out the number of sectors (i.e., find the length number and reconvert to hex). Having done this, you can then use the sector editor to view what is stored in these sectors. This will be data only. If it is text data, you will see the ASCII letters appear along with the hex numbers. If it is not text data, then it will be much harder to interpret.

If you want to know more about how numbers and BASIC programs are filed, then you should consult Mike James's excellent book – *The BBC Micro: An Expert Guide* (Granada).

AMCOM sector changing

The *EDIT action in the Watford utility disk provides for the actions of reading a sector into memory, and reviewing or changing each byte. To perform a similar action with the PACE/AMCOM DFS requires a different set of options. These make use of loading sectors from the disk into memory, searching and editing memory, and saving sectors back from memory. This is the same action as the Watford or any other 'disk-doctor' type of program will perform, but split into separate menu options.

The load sectors option, 1 on the menu, prompts for a starting sector, a starting address, and a number of sectors. These need some consideration. The AMCOM utility itself has a starting address of &13∅∅ and uses 8 sectors, so that its finishing address is 1AFF. This would make the first available free byte at address &1B∅∅. In the example of the use of sector loading in Chapter 5, I made certain of avoiding problems by using &1C∅∅ as a starting address for loading bytes. Unless you are very pressed for space, this is safer, and loading &4∅ bytes will occupy space up to &5C∅∅. If you only want to work on the catalogue, you need only load eight sectors (numbers ∅ to 7) for the AMCOM system, or sectors ∅ and 1 for the Acorn system.

Having loaded the sectors, which is a very rapid process, you can then search and edit the bytes in memory. Unless you save back changed bytes to the disk, you will not change the disk in any way, so that a deliberate effort is needed to make such changes. If you are looking for a catalogue entry, then the 'search memory' option, menu item 6 of the UTILITY menu, is ideal. This prompts for a start address (perhaps &1CØØ) and a stop address (perhaps &5CØØ), and then a string to search for. If you know a title for a file, for example, you can enter it, or the first few letters, and the memory address number will be indicated. If you are looking for a catalogue for a filename, this is a very rapid method. Even if you are looking for a deleted name, it can speed things up if you know the name of the file that was next to the deleted name. Once you find the name of a file, in any of the systems we have examined, it's a straightforward matter to find the file data, such as starting sector number and length.

The menu option 7 then allows you to edit memory. In this option, you can examine memory, byte by byte, from a given starting address. Pressing RETURN brings up another byte, but typing a different valid hex number, then pressing RETURN will change the stored byte. On returning to the menu, option 2 allows you to save memory to the disk. You can then enter the start address, end address and start sector numbers again, and pressing RETURN will copy the memory back to the disk. If you have made any changes, these changes will be copied back on to the disk, and will influence the way the disk behaves. Obviously, the copying action will *not* take place if the disk is write-protected, however.

Disk surgery

A lot of the actions that you can perform with the aid of disk sector editing (meaning changing bytes) can be carried out with the normal DFS commands. It is pointless, therefore, to look at actions like changing filenames or disk titles, since there are commands to carry out these actions. What we particularly need to look at are how to renew a deleted program, and how to combine two files. These appear to be the two features of disk operation that are most vitally needed by users of the disk system.

We'll start with replacing a deleted file. As I noted earlier, the file itself is not deleted by any of the normal deleting procedures (but it *is* deleted by FORMAT). What happens is that the file's catalogue entry is deleted, and all the catalogue entries are shuffled up. We can't, therefore, simply look for a gap in the catalogue! If nothing else is recorded over the file, though, it should be possible to load the bytes of the file, save it under a new name, and so make a new catalogue entry for it. Here's how.

1. Back up the whole disk on to another, just in case.
2. Use the disk sector editor to find the start of your file. This isn't so simple as it looks. One way is to look at the complete catalogue. You will find gaps in it, when you calculate where each file ends. One of these gaps may be your file, the others will be earlier deletions. Once you have some start sector numbers, you can start looking at these sectors to find which of them contains the start of the file you are looking for.
3. When you find the starting sector number, note it. Go through the file now, or use the information from the catalogue, to find the last sector. Work out how many bytes are in the complete file. This is easy in hex, because there are &1ØØ bytes in each sector so that, for example, 6 sectors have &6ØØ bytes stored.
4. Return to the sector editor, and use it to place all the bytes of the file in memory, starting at some convenient place like &1CØØ. Check, if you feel up to it, that this is the correct file.
5. Now return to the operating system, and *SAVE "RENEW" 1CØØ+LEN. If you saved the bytes to 1CØØ, and LEN is the number of bytes in the file (in hex), then this will save the file. The new filename will be RENEW, it will be correctly placed in the catalogue, and it should contain all the bytes of your formerly deleted file. Check that the file works. If, for example, it is a text file, load it in and check it. If it is a BASIC program, RUN it, and so on. Don't take it for granted that you have got it back in one piece! If you have, however, make another copy, under the original name, on the back-up disk. Now compact the disk which had the deleted file.

Merging files

The procedure for merging files is simpler. Suppose that you have two files, TEXT1 and TEXT2. I have chosen this example because text files are the ones that you will most likely want to merge, and because I have had more experience of merging such files! The procedure is, once again, to get the catalogue information for the files. Find the start sector and length of each file. If the files are in adjoining sectors of the disk, merging is very simple. All you need to do is to alter the length bytes in the first file! This leaves the second catalogue entry, but reading the first file name will result in both files being loaded.

The procedure is more difficult if you find that the files are stored at very different parts of the disk. In such a case, copy all the other files on to another disk, and compact the disk with the files you want to merge. The alternative is to copy only the files that you want to merge on to another disk. This will leave the files adjacent, and they can be merged by fiddling with the length bytes in the catalogue of the first file. In detail, this amounts to the following:

1. Note the length of the first file and the length of the second file.
2. Add these numbers.
3. Use the disk sector editor to put this sum number into the length bytes of the first file.
4. If this file is not too long to load into the machine, load it to check.

Finally, remember that disk sector editing can be a very frustrating business. Never, whatever you do, work on any disk for which you have no back-up. Always ensure that your utilities disk is write-protected – it's often a good idea to write-protect your back-up disk as well, particularly if you have more than one drive. It's surprising how many times in the small hours of the morning, a write that you thought was to Drive 1 goes to Drive Ø ...

Chapter Seven
BASIC Filing Techniques

What is a file?

I have used the word 'file' many times in the course of this book to mean a collection of information which we can record on a disk. We have used, for example, BASIC files (programs), machine code files (also programs), text files (such as WORDWISE text) as examples in previous chapters. In this chapter, I shall use the word 'file' in a narrower sense. I'll take it to mean a collection of data that is separate from a program. For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the action of the program, and it preserves these amounts for the next time that you use the program. Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings. This information is a file and, at some stage in the program, you would have to record this file. Why? Because when you load a BASIC program and RUN it, it starts from scratch. All the information that you fed into it the last time you used it has gone – unless you recorded that information separately. This is the topic that we're dealing with in this chapter – recording the information that a program uses. The shorter word is *filing* the information.

You can't discuss filing without coming across some words which are always used in connection with filing. The most important of these words are *record* and *field*. A 'record' is a set of facts about one item in the file. For example, if you have a file about LNER steam locomotives, one of your records might be used for each locomotive type. Within that record, you might have the designer's name, firebox area, working steam pressure, tractive force . . . and anything else that's relevant. Each of these items is a 'field', an item of the group that makes up a record. Your record might, for example, be the Scott class 4-4-0 locomotives. Every different bit of information about the Scott class is a field, the whole set of fields is a record, and the Scott class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purposes locos, and so on. Take another example, the file 'British Motorbikes'. In this file, B.S.A. is one record, A. J. S. is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, suspension, top speed, acceleration ... and whatever else you want to take note of. Filing is fun – if you like arranging things in the right order.

Disk filing

In this book, because we are dealing with the BBC Disk systems, we'll ignore filing methods that are based on DATA lines in a BASIC program, or on the use of cassettes. Though you may be experienced with the use of filing with cassette systems, I'll explain filing from scratch in this chapter. This is because many buyers of the BBC machine nowadays start from scratch with a disk system, and have never used cassettes. If it's all familiar to you, please bear with me until I come to something that you haven't met before.

To start with, there are two types of files that we can use with a disk system, *serial files* and *random access files*. The difference is a simple, but important one. A serial (or sequential) file places all the information on a disk in the order in which the information is received, just as it would be placed on a cassette. If you want to get at one item, you have to read all of the items from the beginning of the file into the computer, and then select. There is no way in which you can command the system to read just one record or one field. A random access file does what its name suggests – it allows you to get from the disk one selected record or field without reading every other one from the start of the file. You might imagine that, faced with this choice, no-one would want to use anything but random access files. It's not so simple as that, though, because the convenience of random access filing has to be paid for by a lot more complication, as we'll see. We'll start, then, by looking at serial files, which are also easy to record on cassette. All of the DFS commands for serial filing are the same as the commands of the cassette filing system. This makes the change very easy if you have been using filing on cassette and you then upgrade to disk. If you have never used cassette files, of course, it's all new.

Serial filing on disk

We'll start by supposing that we have a file to record, called CAMERAS. On this file we have records (such as Nikon, Pentax, Canon, Yashica, and so on). For each record we have fields like film size, shutter speed range, aperture range (standard lens), manual or automatic, and so on. How do we write these records? First of all, we need to arrange the program that has created the records so that it can output them in some order. The usual order will be to take the records in some chosen order, and output the fields of the record in some order as well. BBC BASIC provides well for this by the use of FOR ... NEXT or REPEAT ... UNTIL loops. Figure 7.1, for example, shows how we might arrange this part of a BASIC

```
  100 DIM field$(5):X%=0:REPEAT:CLS:PRIN
T" Type X to end entry"
  110    INPUT"Record name ",rec$:X%=X%+1
  120    REM Need to record now!
  130    FOR N%=1TO5
  140      PRINT"Field item ";N%;"  ";:INP
UT field$(N%)
  150      REM Need to record this one al
so!
  160      NEXT
  170    UNTIL rec$="X":X%=X%-1
  180 REM Leaves a blank record at the e
nd of the file.
  190 PRINT"There are ";X%;" records on
the file."
```

Fig. 7.1. The BASIC loops which will be used to create a file, with several fields to each record.

program so as to output a number of records, with five fields to each record. The number of fields is five, so the fields are put out using a FOR N%=1 TO 5 loop. The number of records isn't fixed, so we use a REPEAT ... UNTIL loop, which keeps putting out records until it finds one called "X", which is the terminator. Note that we haven't used an array for holding these items, because an array has to be dimensioned, and we don't know in advance how many items we will have.

That deals with the organisation of the data for putting on to disk, but how do we actually put it on the disk? There are several stages, and the first one is to allocate a *channel number*. This is a type of code that the machine will use to distinguish files. The BBC machine can deal with several sets of files at one time, five to be precise. It's most unlikely that you will ever want to use more than two files at a time (probably one for reading and one for writing, for example), but it's better to be generous than to be stingy. Each time you want to make use of a file, then, you must have a channel number (or 'handle') allocated. The machine does this automatically, and the keywords that are used in this connection are OPENIN, OPENUP and OPENOUT.

Now, at this point, things can get confused. At the time of writing, there are two versions of BASIC used in the BBC machine, BASIC I and BASIC II. To find out which one you have, type REPORT and press RETURN. If you get a copyright notice with the date 1982 on it, then your BASIC version is BASIC II. If an earlier date appears (usually 1981), then you have BASIC I. There are two important differences. One is that BASIC I does *not* have the OPENUP command. The other is that a filing program which has been written on a machine equipped with BASIC II and which makes use of OPENIN, may not run correctly on a machine which uses BASIC I. This is not a problem provided that you are aware of it, but it can cause a lot of head-scratching if you are not. Figure 7.2 summarises what these commands are used for in the two versions of BASIC. For the purposes of this chapter, assuming that most new users of disk machines will have BASIC II, I shall use the commands that are available with BASIC II.

After that short diversion, let's go back to our filing program.

BASIC I

OPENIN will open a file for reading, and also for writing.
OPENOUT will open a file for writing only.

BASIC II

OPENIN will open a file for reading only.
OPENOUT will open a file for writing only.
OPENUP will open a file for reading or writing.

If you are writing programs on a BASIC II machine which you will want to use on a BASIC I machine, use OPENOUT and OPENUP, not OPENIN. If you have a program which was written on a BASIC I machine, and you want to run it on a BASIC II machine, replace each OPENIN with OPENUP.

Fig. 7.2. The use of OPENIN, OPENUP, and OPENOUT.

Before we start to gather the data together for filing, we need to 'open a channel' for the data. In either version of BASIC, this can be done using the OPENOUT command. For example:

```
2ØØØ nr%=OPENOUT"CAMERAS"
```

will open a file called CAMERAS for writing (that's why we used OPENOUT), and allocates the channel number that OPENOUT creates to the variable nr%. This channel number will be referred to each time we record data. If open another file for writing, it will be allocated another channel number, so that we can keep the files separate. To avoid confusing yourself, though, try to keep as few files on the go as possible. The channel number is used to locate the address of the piece of RAM memory which is used as a buffer to store data before recording or after replaying.

The use of the OPENOUT command opens a file – which means that we can make use of the file. It also means that the disk is prepared for the file. Any file that exists on the disk already and has the same name of CAMERAS will be deleted, unless it is locked. In addition, sectors will be reserved for the file. If you are using a new disk, 64 (denary) sectors will be allocated for use with this file. If your disk had a file called CAMERAS which was deleted by the action of OPENOUT, then only the number of sectors that was used by the old file will be allocated to the new one. This is done because, when you replace a file by another of the same name, it's usually because you have made minor changes to the file. It can cause problems, however, if the space lies between two other files on the disk, and is now too small for the altered file. In such a case, it's better to delete the old file before saving the new one. If this has to be done, you should be sure that you have a back-up copy of the file – just in case! The information that the computer writes to the disk at this point is usually called the 'header'. It's the first part of any file, and the data will follow it. If you 'close the file', by typing CLOSE #nr% (then press RETURN), you will have a file that contains a header, and nothing else – an empty file. That's not particularly useful, so we had better see how we can put data into a file.

It's at this stage that we need our REPEAT ... UNTIL and FOR ... NEXT loops. Within these loops, we need to have a line something like:

```
3ØØØ PRINT #nr%,field(N%)
```

PRINT#nr% means put the information out on channel nr%. This, because of our previous OPENOUT, leads to the disk system, so that

PRINT#nr% will eventually put out to the disk system the data that follows. In this example, it's field (N%). N% is the number in the FOR ... NEXT loop, so that as the loop goes round, we will put on to the disk field(1), then field(2), then field(3) ... and so on. We also need to write the 'record' name, and this would have been done within the REPEAT ... UNTIL loop, by using a line such as:

```
2500 PRINT#nr%, record
```

without using an array (because of the unknown amount of dimensioning). Figure 7.3 shows an example of a very short and simple program of this type which uses DATA lines to avoid the waste of time that is incurred if INPUT is used to get a response from you. You can, of course, easily change this program so that it gets

```
   10  ONERROR GOTO 1000
  100  DIM field$(4):nr%=OPENOUT"amplifi
ers":REPEAT
  110   READ record$:IF record$="X"THEN
160
  120   PRINT#nr%,record$
  130   FOR N%=1TO4
  135      READ field$(N%)
  140          PRINT#nr%,field$(N%)
  150     NEXT
  160   UNTIL record$="X": CLOSE#nr%
  170 END
  210 DATACrimson,510/520,42,20-58,-77
  220 DATAJVC,AX1,37,8-71,-87
  230 DATAMarantz,PM5,100,5-51,-82
  240 DATAPioneer,A8,108,0-75,-89
  250 DATASansui,A9,88,7-63,-77
  260 DATAX
 1000 ONERROROFF:CLOSE#nr%:REPORT
```

Fig. 7.3. A file-creating program, showing the structure of the loops and the use of OPEN and CLOSE commands. DATA lines have been used in place of INPUT to make the example run more quickly. The ONERROR lines have been included to make sure that files will be closed if there is an error – or if ESCAPE is pressed.

information from you by way of INPUT. Before we move on, consider what this has done. It has created a file called amplifiers, and allocated a channel number to this file (we could find this channel number by typing PRINT nr%). It has then stored the data as it came along, in the sequence of RECORD, then FIELDS.

Finally, the file has been recorded and closed by using CLOSE #nr%. This last step is very important. For one thing, you don't actually record on the disk any of the information in this short program until the CLOSE#nr% statement is executed. That's because it would be a very time-consuming business to record each item of a file at a time. What the DFS does is, as you might suspect by now, to gather the data together in memory. This is a 'buffer' piece of memory, and it will be written to the disk only under one of two possible circumstances. One is that the buffer is full, so that there is one sector full of data (256 bytes) to write. The other is that there is a CLOSE#channelnumber type of statement in the program. For a large amount of data, the disk will spin and write data each time the buffer is full. When data is read from DATA lines, the buffer fills so fast that you will not notice that separate write operations are involved. The CLOSE# command then writes the last piece of data, the one which doesn't fill the buffer. It also writes a special code number, called the end-of-file code (EOF). This can be used when the file is read, as we'll see later. If you forget the CLOSE# statement, you'll leave the buffer unwritten, with no EOF – and cause a lot of problems in your programs. Forgetting the CLOSE# is called leaving your files open, and you wouldn't like to be seen like that, would you? A useful hint is to include in every disk-filing program an:

```
ONERRORGOTO
```

line which leads to a CLOSE#Ø statement. The effect of CLOSE#Ø is to close all open files, so that if an error (and this includes pressing the ESCAPE key) occurs, your data will be recorded. Incidentally, if you use an INPUT statement to gather up the data, you can find that with a lot of data you will hear the disk start and stop at intervals. That's an indication of the buffer transferring data to the disk. You can't use the keyboard while the transfer taking place is happening, but the time that's needed to write a sector is so short that it's not much of a hazard.

Getting your own back

Having created a file on disk, we need to prove that it has actually happened by reading the file back. A program which reads a file must have, early on, a command which opens the file for reading. This can be OPENIN or OPENUP. In BASIC II, OPENIN will open a file for reading only, not for writing. In BASIC I, OPENIN opens a file either for reading or for writing. At this point, we'll stick to OPENIN. The purpose of OPENIN, like OPENOUT, is to open a file and allocate another channel number. This need not be the same as the one that was used to write the file, but it must have the same name. If we recorded a file using OPENOUT"CAMERAS", then we must not expect to be able to read it if we use OPENIN "CAREMAS" – or any other name. Misspelling can haunt you here! Once the channel number has been allocated, we can read data with INPUT#nr%. This reads an item from the disk, and will allocate it to a variable name or print the item, according to what we have programmed. The number of reads can be controlled by a FOR ... NEXT loop, if the number is known, or it can make use of EOF, if the number is unknown. The example of Fig.7.4 shows both methods in use. The number of fields has been four, so that a FOR

```
   10 ONERROR GOTO 1000
  100  DIM attribute$(4):nr%=OPENIN"ampl
ifiers":REPEAT
  110    INPUT#nr%,name$
  120    PRINT"Makers Name- ";name$
  130    FORN%=1TO4:INPUT#nr%,attribute$(
N%):NEXT
  140        PRINT"Model  - ";attribute$(1)
  150    PRINT"Max. Power (Watts) - ";att
ribute$(2)
  160        PRINT"Bandwidth (Hz - KHz)- "
;attribute$(3)
  170    PRINT"Noise at zero volume (db.)
 - ";attribute$(4)
  180    PRINT:PRINT"Press any key to con
tinue"
  190     A$=GET$
  200        UNTIL EOF#nr%
  210 CLOSE#nr%
  220 END
 1000 ONERROROFF:CLOSE#nr%:REPORT
```

Fig. 7.4. Reading a file. The FOR ... NEXT loop can be set up to read the known number of fields, but the unknown number of records is dealt with by searching for the EOF marker.

... NEXT loop can be used to control the input of the fields. The number of records, however, has not been settled by a FOR ... NEXT loop, so that we have to keep reading the file until the EOF byte is found. The BBC BASIC allows us to use a REPEAT...UNTIL loop for this, with UNTIL EOF#nr% used to stop the reading process. Note that the disk does not spin each time you press a key to get another record. This is because a complete sector is read each time, and if the information that you want is all on the same sector, the disk need not be used.

Now, this simple example shows a lot about serial filing that you need to know. When you use disks, then the name that is used with OPENOUT is the filename for the file on the disk. Any other file that is later recorded with the same name will overwrite this file. This is an important point to emphasise if you have been using cassettes, because you have more control over a cassette. You can write a file called INDEX at the start of a tape, for example, then wind the tape on slightly and record another, different, file with the same name. You certainly can't record two files with identical names on one disk. In addition, a file is closed by writing the EOF character. How, then, can you update a file, particularly if you want to add more items to the end of the file?

There are two answers, if we stick to serial filing. One possibility, which is the simplest one for short files, is to load the whole file into the memory of the computer, make the alterations (your BASIC program will have to be written so as to provide for this), and then write the file again, wiping out the earlier version. The other possibility is to open two files, one for reading and the other for writing. You don't need to have dual disk drives for this, though it makes life much simpler if you do. This means that the computer will maintain two buffers. You read one record from the reading file and display it. If it's all right, it's then written (to the buffer initially). If the record has to be modified, you can do so. If extra records have to be added, this is equally simple. Each time a buffer empties, the disk will spin and a read or write will take place. This 'simultaneous' operation is possible because of the use of different channel numbers, which control different buffers. In practice, it's a matter of writing your program to suit. Figure 7.5 shows a simple program which allows you to extend the file that was created by the program of Fig.7.3. Note, however, that the files use different names, because I have assumed that both files will be on the same disk. We must, therefore, end the program by deleting the old file and changing the name of the newly created file (the extended or changed file) so that it has the same name as the old file. This can be done from BASIC by using *DELETE and *RENAME just as we would use them direct from the keyboard. Note carefully, if you are addicted to the use of

```
  10 ONERROR GOTO 1000
 100  DIM attribute$(4),field$(4)
 120 nr%=OPENUP"amplifiers":new%=OPENOU
T"moreamplifiers"
 130 REPEAT
 140    INPUT#nr%,name$:PRINT#new%,name$
 150    FOR N%=1TO4:INPUT#nr%,attribute$
(N%):PRINT#new%,attribute$(N%):NEXT
 160    UNTIL EOF#nr%
 170 CLOSE#nr%
 180 REPEAT:CLS:PRINT"Enter X to end ad
ditions."
 190    INPUT "Makers Name - ";name$
 200    IF name$="X" THEN260:ELSE PRINT#
new%,name$
 210    RESTORE:FOR N%=1TO4
 220        READ field$(N%):PRINT field$(
N%);"- ";:INPUT attribute$(N%)
 230       PRINT#new%,attribute$(N%)
 240       NEXT
 250    UNTIL FALSE
 260 CLOSE#new%
 270 *DELETE amplifiers
 280 *RENAME moreamplifiers amplifiers
 290 END
 500 DATAModel,Power (Watts),Bandwidth
(Hz-kHz.),Noise at zero volume
1000 ONERROROFF:CLOSE#0:REPORT
```

Fig. 7.5. Extending a serial file. This illustrates the routines that are needed for extending a file, particularly when only one drive can be used.

multistatement lines, that you cannot have another command following a star-command (like *DELETE or *RENAME) in the same line. Note also the use of CLOSE#∅ in line 1∅∅∅. This will close any open file regardless of its channel number. We need this command, because we can't be sure which channel might be open in the event of an error.

With two drives available, you can use the type of program that is illustrated in Fig. 7.6. The OPENUP and OPENOUT commands now use a full filename, containing directory and drive information, and this allows you to use the same filename for the data. If you make use of this method, though, you have to be sure that you know which disk contains the up-to-date file!

```
   10 ONERROR GOTO 1000
  100  DIM attribute$(4),field$(4)
  120 nr%=OPENUP":0.$.amplifiers":new%=O
PENOUT":1.$.amplifiers"
  130 REPEAT
  140    INPUT#nr%,name$:PRINT#new%,name$
  150    FOR N%=1TO4:INPUT#nr%,attribute$
(N%):PRINT#new%,attribute$(N%):NEXT
  160    UNTIL EOF#nr%
  170 CLOSE#nr%
  180 REPEAT:CLS:PRINT"Enter X to end ad
ditions."
  190    INPUT "Makers Name - ";name$
  200    IF name$="X" THEN260:ELSE PRINT#
new%,name$
  210    RESTORE:FOR N%=1TO4
  220        READ field$(N%):PRINT field$(
N%);"- ";:INPUT attribute$(N%)
  230       PRINT#new%,attribute$(N%)
  240       NEXT
  250    UNTIL FALSE
  260 CLOSE#new%
  290 END
  500 DATAModel,Power (Watts),Bandwidth
 (Hz-kHz.),Noise at zero volume
 1000 ONERROROFF:CLOSE#0:REPORT
```

Fig. 7.6. How a serial file can be extended when two drives are in use. The OPEN commands use the drive numbers to ensure that the file is read from one drive and written on the other.

Random access filing

Random access filing, as the name suggests, means that we should be able to get at any record (or set of records) on the disk *without* having to read all of the earlier records as well. This doesn't mean that we have to create a random access file by scattering bytes all over the disk (though we could); it simply means that it should be possible to locate and read any byte on the disk. In general, we will create a file initially, and record it, using just the same techniques as we have used for serial filing. We can then add to the file, replace items, change items, and select items at random, using random access techniques. The important point about such random access filing is that it's possible to extend a file so that the file length is much greater than could be held in the memory of the computer. As an alternative, of course, we can use the random access filing methods that we need for extending a file in order to create the file in the first place.

Random access commands

The important random access commands of the DFS are PTR#, EXT#, BGET#, and BPUT#, in addition to the filing commands that you have met already. Of these, the one that needs most explanation is PTR#. This has to be followed by a channel number for an open file, and its action is hinted at by the full version of its name – pointer. The pointer is a number which refers to a byte on the disk. The numbering system is simple – the bytes of data in the file are simply numbered in order, starting with Ø at the beginning of the file. When you open a file, the value of PTR#nr% (assuming you are using channel number nr%) will be zero. By changing the value of PTR#nr%, you can read or write a byte at that number. The actual manipulation of bytes is done in a memory buffer, one sector at a time, and bytes are saved to or loaded from the disk also one sector at a time. It all sounds very simple, but how do you know which number to use? The answer depends very much on what you intend to store on the disk.

Integer files

Suppose that what you are filing is a collection of money amounts. Now a money amount such as £231.42 can be expressed as an integer by converting to pennies, an amount of 23142. The BBC Micro permits an unusually large range of integer numbers, up to more than two thousand million, so that unless you are keeping books for a large corporation, you won't find this a restriction. Many computers can use integers only up to the number 32767.

Now the point of talking about integers is that the BBC machine stores its integers on disk in a fixed form. The first byte that is put on the disk is &4Ø, which is used as a code to indicate the use of an integer. This appears as the symbol @ in an ASCII printout. The number itself is coded as four more bytes, making five in all. If you store ten integer numbers on a disk, then, you know that the first number is stored at PTR position ∅, the next one at PTR position 5, the next one at PTR position 1∅, and so on. If you want the sixteenth number, then the pointer has to be set to 5×15=75. The formula for the pointer number is (data number −1)×5, where 'data number' means the number of the item (first, second, third and so on). We can therefore create a random access file by placing the pointer correctly, then using PRINT#nr% to write the number. This can be done in a loop for writing, placing the numbers in serial form. For reading a file, however, we need only to choose a number, set the PTR# value, and then use INPUT# for that one item. The simple program of Fig. 7.7 illustrates this technique. I have used 21−N% so that you can

```
   10 ONERRORGOTO 1000
  100 DIMA%(20):nr%=OPENOUT"integers"
  110 FOR N%=1TO20
  120    A%(N%)=(21-N%):NEXT
  130 FOR N%=1TO20
  140    PTR#nr%=(N%-1)*5
  150    PRINT#nr%,A%(N%):NEXT
  160 CLOSE#nr%
  170 rd%=OPENUP"integers":REPEAT
  180    INPUT"Pick an item between 1 and
 20",select%
  190    IF select%<1 OR select%>20 THEN
PRINT"Incorrect selection, please try ag
ain":GOTO180
  200    PTR#rd%=(select%-1)*5
  210    INPUT#rd%,X%:PRINT"Iten ";select
%;" is ";X%
  220    UNTIL FALSE
 1000 ONERROROFF:CLOSE#0:REPORT
```

Fig. 7.7. A random access file of integers. This makes use of the PTR command to find a place in the file. This is actually done in the buffer memory, with the file being read one sector at a time.

check that the numbers which are returned are correct. This is not possible if you use random numbers for checking, and you need to be slightly suspicious of ascending numbers in case they are related to a loop count!

This same scheme can be used also for 'real' numbers. Real numbers are coded differently, and their use can cause approximations which are never needed when you deal with integers. For a very large number range (up to 10^{38}) and when we need negative numbers and fractions, however, reals have to be used. You can write a file of real numbers and read it back using exactly the same techniques as you have met for integers. The only differences are that a real number is identified by the code &FF, and uses another 5 bytes for the number value. This makes six bytes in all, so that the formula for setting the pointer is (data number−1)×6. To get the 27th data number, then, you set PTR# to the number 26 × 6 = 156.

Random file-random length

Random filing is simple if you deal with one type of number only. What happens, though, if you need to record a mixture of real and integer numbers? Worse still, what happens if you need to record a set of strings? A string can have a length which ranges from zero (null string) to 255 characters. Obviously, we can't simply apply a formula to any file that uses strings or mixed characters. This is where the headaches start – but, fortunately, there's a wealth of experience to draw upon. Large mainframe computer programmers were dealing with random access files long before microcomputers were dreamt of, and the techniques for dealing with these problems were also developed before many of today's programmers were born. In this chapter, I can do no more than lift a corner of the stone of accumulated wisdom. For more detail on filing techniques, you will have to consult a book that deals in much more detail with this subject. The principles, however, can be explained here. We'll start with the idea that we shall deal with mixtures of reals, integers and strings simply by converting everything to strings. In any database program that requires you to input information (where else would it get the information?), the use of INPUT can be to a string variable. Any number that is entered will therefore be converted to string form. If you have a number (such as a FOR ... NEXT loop count number) which is in integer or real form, it can be converted to string form by the use of STR$. The problem then boils down to how you record and replay strings of different lengths, using random access.

String variable storage

First we need to know how the BBC machine stores string variables on a disk. The first byte that is recorded is &ØØ, which is the 'marker' for a string. The second byte is the number that gives the length of the string. The characters of the string are then placed on the disk *in reverse order*. The point about reverse order is important to note, in case you are reading the string by use of a disk sector utility. Note that this happens only if the string has been placed using PRINT# – there are other methods, as we shall see. When we are faced with the problem of writing strings of various lengths, there are two ways of tackling the problem. One is to make each string of the same length. This is done by fixing a maximum length for the string, and making each string fit. Longer strings are simply chopped to size, shorter strings are padded with blanks, so that all reach the same length. It's a simple technique, which is used to a large extent. If you stay in a village with a name like LITTLEPUDDLINGTON PARVA, you may find that computer-addressed mail (like all those encyclopaedia offers) will usually miss out the last letters.

The alternative is much harder. You will have to locate each string, find its length from the length byte, add this to the PTR# number and continue until you have found the 54th string, or whatever you are looking for. This is a clumsy method, because it means that the disk is being used continually just to pick up single bytes. This takes as long as it would to read the whole record, so it's not really a viable alternative. It can lead to a viable alternative, however, in the form of a filing method that is called *indexed sequential*. This combines the best features of both serial and random access files – but there will be more about it later.

For the moment, we'll stick to the methods of string filing that depend on padding or cutting to a fixed length. Suppose, for example, that we want to create a file called COUNTRIES. Each record would consist of a country name, and the fields would consist of the name of the capital city, the name of the principal unit of currency, and the population at the last census. It's a simple example, and we'll make extensive use of it in the remainder of this chapter.

To start with, we have to choose field lengths. For the record field (the name of the country) a maximum of twenty characters seems reasonable. Most countries have reasonably short names, but we need space for those which have the word 'Republic' as part of the name. We may have to abbreviate this to Rep. in some cases. Having fixed the length of the record as twenty characters, we can now look at the fields. Field 1 is the capital city, and a length of twenty looks reasonable for this as well. Field 2 is the currency field, and ten characters seems adequate, if we stick to the generally used names. For example, we'll use dollar for the currency name of Trinidad and

Tobago, rather than the full name of Trinidad and Tobago Dollar! This might seem a nit-picking exercise, but it's typical of the type of decisions that you have to take when you are deciding on field lengths. Finally, for Field 3, population, we'll use six characters. Just six? Yes, because we can count population in thousands – it saves three characters.

The next step is to choose variable names. Since the BBC machine allows us to use long variable names, we can take advantage of this by using country$ for the country name, capital$ for the capital, coin$ for currency and pop$ for population. That's clear without needing too much typing! The next thing that we have to worry about is how we ensure that each string is cut or padded to the right size. The method is simple enough. What we have to do is to add spaces to the end of each string, then take the left-hand side for as many characters as we need. Because we have to deal with strings of different names and with different field sizes, this calls for either a procedure or a defined function. Figure 7.8 shows examples of each – take your pick, but note that you have to be careful about what names you use in the procedure.

```
   10 REM
   20 REPEAT
   30    INPUT "Name",Name$
   40    INPUT "Pad length",length%
   50    PROCpad(Name$,length%)
   60    PRINT Name$:PRINT LEN(Name$)
   70    PRINTFNpad(Name$,length%):PRINT
LEN(FNpad(Name$,length%))
   80    UNTIL FALSE
  100 END
  200 DEFFNpad(Name$,length%)=LEFT$(Name
$+STRING$(length%," "),length%)
 1000 DEFPROCpad(a$,b)
 1010 Name$=LEFT$(a$+STRING$(b," "),b)
 1020 ENDPROC
```

Fig. 7.8 Using a defined function or a procedure to pad a string to length. The program includes an example, allowing you to test the technique.

Having got each record and field of the correct length, what then? One possibility is to record them separately. This, however, can be wasteful, though if the strings are long it might be necessary. Remember that to record a string, you need to place two other bytes on the disk, the identifier byte (&FF) and the string length byte. For the four strings that we are dealing with here, that's a total of eight bytes added to the 56 characters that we want to place on the disk for each record. For a file of 100 entries, we will be wasting 800 bytes. In this example, it's not very significant, but for other files it would certainly be. Suppose you had twenty fields, each quite short, and you needed 5000 records? The waste would be very considerable, so it's worth eliminating. The simplest way of eliminating the waste is to record one string, not four. By joining the strings into one, we then need only two 'wasted' bytes per record instead of eight. We can still separate the strings just as easily when the large record string is read back, because each string is of fixed length. Figure 7.9 shows how the strings are joined and how they are separated again.

```
   10 a$="":FOR J=1TO5
   20    INPUT"Name - ",Name$
   30    a$=a$+FNpad(Name$,15)
   40    NEXT
   50 PRINT a$
   60 FOR X=1TO5
   70    PRINT MID$(a$,15*X-14,15)
   80    NEXT
  100 END
 1000 DEFFNpad(N$,N)=LEFT$(N$+STRING$(N,
" "),N)
```

Fig. 7.9. Combining strings into a large string, and separating them again. This is simple because the fields are of fixed length.

Finally, we need to look at how we record and replay the records. Since each file has a fixed length of 58 bytes, and will have two bytes added by the DFS, we can make up a formula for the PTR# number. If we start with PTR# zero, then the formula for the Nth string will be (data number − 1) × 58. By placing the pointer at the correct position and using PRINT#, we can put the strings on to the disk. By using the formula to find the PTR# number for a given number of record, such as the 24th, we can place the pointer at the start of a string (meaning the &FF byte) and then use INPUT# to read the complete string. That, in essence, is it! Now we need to look at the actual program example, and try it out.

The COUNTRIES FILE

Figure 7.10 shows the complete COUNTRIES FILE program. In this

```
   10 REM COUNTRIES FILE
   20 REM
   30 REM
   40 REM
   50 REM Ian Sinclair 1983
   60 REM
   70 REM
   80 ONERRORGOTO1410
   90 n%=OPENOUT"Countryfile"
  100 FOR N%=1TO10:record$=""
  110    READ country$,capital$,coin$,pop
$
  120    record$=FNpad(country$,20)+FNpad
(capital$,20)+FNpad(coin$,10)+FNpad(pop$
,6)
  130    PTR#n%=(N%-1)*58
  140    PROCrecord
  150    NEXT
  160 CLOSE#n%
  170 CLS:PRINT"Your skeleton file is co
mplete."
  180 PRINT"You can now _ "
  190 n%=OPENUP"Countryfile"
  200 PROCmenu("Add record","Change reco
rd","Read record","END")
  210 IF choice%=1 THEN PROCadd
  220 IF choice%=2 THEN PROCchange
  230 IF choice%=3 THEN PROCread
  240 IF choice%=4 THEN 280
  250 CLS:PRINT"Would you like to return
 to the menu?":PRINT"Please type Y or N"
  260 IF choice%=4 THEN 280
  270 A$=GET$:IF A$="Y" OR A$="y"THEN 20
0
  280 CLOSE#0
  290 PRINT"End of Program.":END
  300 DEFPROCadd
  310 PROCend:REPEAT
  320    PROCgetdata
  330    PROCrecord
  340    PROCupdate
  350    PROCspacebar
  360    UNTIL A%<>32
  370 CLOSE#0:n%=OPENUP"Countryfile"
  380 ENDPROC
  390 DEFPROCchange:REPEAT
  400   PROCfind
  410   PROCprintdata
  420   PROCchangedata
  430   PTR#n%=position%
  440   PROCrecord
  450   PROCspacebar
  460   UNTIL A%<>32
  470 ENDPROC
  480 DEFPROCread:REPEAT
  490   PROCfind
  500   PROCprintdata
  510   PROCspacebar
  520   UNTIL A%<>32
  530 ENDPROC
  540 DEFPROCend
  550 PTR#n%=EXT#n%:point%=PTR#n%:ENDPRO
C
  560 DEFPROCgetdata
  570 CLS:PRINT'';TAB(17)"DATA"
  580 PROCcountry:PROCcapital:PROCcoin:P
ROCpop
  590 PROCpack
  600 ENDPROC
  610 DEFPROCcountry
  620 INPUT'"Name of country- ",a$:count
ry$=FNpad(a$,20):ENDPROC
  630 DEFPROCcapital
  640 INPUT'"Name of capital- ";a$:capit
al$=FNpad(a$,20):ENDPROC
  650 DEFPROCcoin
  660 INPUT'"Name of currency- ";a$:coin
$=FNpad(a$,10):ENDPROC
  670 DEFPROCpop
  680 INPUT'"Size of population, in thou
sands- ";a$:pop$=FNpad(a$,6)
  690 ENDPROC
  700 DEFPROCrecord
  710 PRINT#n%,record$
  720 ENDPROC
  730 DEFPROCupdate
  750 PTR#n%=point%+58
  760 ENDPROC
  770 DEFPROCspacebar
  780 PRINT''"Press spacebar to continue
```

```
.":PRINT"Press any letter key to stop."
  790 A%=GET:ENDPROC
  800 DEFPROCfind
  810 PRINT"Please type number, then RET
URN"
  820 INPUT'"Item number, please- ",sear
ch%:CLS
  830 position%=(search%-1)*58
  840 IF position%>=EXT#n% THEN PRINT"No
t on file.":PRINT"Please try a lower num
ber":GOTO820
  850 PTR#n%=position%:INPUT#n%,record$
  860 ENDPROC
  870 DEFPROCprintdata
  880 CLS:PRINT'';TAB(17)"DATA"''
  890 country$=LEFT$(record$,20)
  900 capital$=MID$(record$,21,20)
  910 coin$=MID$(record$,41,10)
  920 pop$=RIGHT$(record$,6)
  930 PRINT"COUNTRY - ";country$
  940 PRINT'"Capital city is - ";capital
$
  950 PRINT'"Currency is the - ";coin$
  960 PRINT'"Population (in thousands) i
s- ";pop$
  970 ENDPROC
  980 DEFPROCchangedata
  990 PRINT'"Which one do you want to ch
ange?"
 1000 PROCmenu("Country","Capital","Curr
ency","Population")
 1010 IF choice%=1 THEN PROCcountry
 1020 IF choice%=2 THEN PROCcapital
 1030 IF choice%=3 THEN PROCcoin
 1040 IF choice%=4 THEN PROCpop
 1050 choice%=0:PROCpack
 1060 ENDPROC
 1070 DEFPROCpack:record$=""
 1080 record$=country$+capital$+coin$+po
p$
 1090 ENDPROC
 1100 DEFPROCpoint
 1110 PTR#n%=point%
 1120 ENDPROC
 1130 DEFPROCrecord
 1140 PRINT#n%,record%
 1150 ENDPROC
 1160 DEFPROCmenu(a$,b$,c$,d$)
 1170 PRINT';TAB(17)"MENU"'
 1180 PRINT"1.  ";a$;"."
 1190 PRINT"2.  ";b$;"."
 1200 PRINT"3.  ";c$;"."
 1210 PRINT"4.  ";d$;"."
 1220 PRINT'"Please select by number-":P
RINTTAB(3)"Do not use RETURN key."
 1230 PROCchoice
 1240 ENDPROC
 1250 DEFPROCchoice
 1260 A=GET:choice=A-48
 1270 IF choice<1ORchoice>4 THEN PRINT"I
ncorrect - 1 to 4 only.":PRINT"Please tr
y again.":PROCchoice
 1280 choice%=choice
 1290 ENDPROC
 1300 DEFFNpad(d$,e%)=LEFT$(d$+STRING$(e
%," "),e%)
 1310 DATAAustria,Vienna,Schilling,7521
 1320 DATABrazil,Brasilia,Cruzeiro,10170
7
 1330 DATAChile,Santiago,Escudo,10229
 1340 DATADenmark,Copenhagen,Krone,5025
 1350 DATAEgypt,Cairo,Pound,35619
 1360 DATAFinland,Helsinki,Markka,4668
 1370 DATAGreece,Athens,Drachma,8950
 1380 DATAHonduras,Tegucigalpa,Lampira,2
781
 1390 DATAIndia,Delhi,Rupee,574216
 1400 DATAJapan,Tokyo,Yen,108346
 1410 ONERROROFF:CLOSE#0:REPORT:PRINT" I
n line ";ERL:END
```

Fig. 7.10. The COUNTRIES FILE program. This is a rather long example, but it illustrates a large number of techniques in a way that short examples fail to do.

example, I have used READ ... DATA lines in place of INPUT lines to place the first set of data into the file. This is because it saves effort on your part in creating the file initially. Once the file has been created, however, you can add to it and alter it as you like. The important point, though, is to realise why each step is taken, so we'll look at the program now in detail. The lines have been numbered in tens so as to make entry with the AUTO facility of the BBC machine easy.

This, I must point out, is not intended to be any sort of object-lesson in how to write filing programs. If you want that, you should look at the disk version of the MASTERFILE program by Sheridan Williams, available from the BEEBUG User Group. What I have set out to do is to illustrate, in a relatively simple but reasonably useful example, what can be done. It's more important that it should give you ideas, and show how problems can be avoided, than that it should be a useful program in its own right. To start with, then, lines 9ϕ to 16ϕ open a file for writing, and place some information into it. If you have a program which allows the choice of reading a file, you have to make sure that there is a file to read. These lines provide one. If these lines did not exist, you would have to test whether a file existed. This is done by opening for reading and testing the channel number. If this is zero, then no file exists. You can test the channel number so as to print a warning, and then move to another part of the menu. Another option is to open a file for writing and close it again. This will be a blank file, but at least it can be opened for reading even though there is nothing in it!

Ten of these 'sample' records are put on to the disk, using the techniques that we have dealt with earlier in this chapter. The names are padded to length, run together, then recorded, and then the file is closed. The fun starts at line 17ϕ. This introduces a menu (in line 2ϕϕ) which allows you to add to the file, change items, read items, or just end it all. All of these choices have something useful to say about file handling. We'll start with the simplest option, END. This leads, in line 24ϕ, to line 28ϕ (sorry about that GOTO, Roy), which closes all open files (hence the use of CLOSE#ϕ). This must be done before leaving any filing program. Line 19ϕ also prints a message because, if you use END alone, all you get is the > prompt, leaving you wondering if the program has ended or not.

There's rather more to option 3, reading files. The procedure starts in line 48ϕ. The REPEAT means that more than one file can be read, and the PROCEDURE involves calling a number of other PROC lines. These, identified by name, are PROCfind, PROC-printdata and PROCspacebar. PROCfind, in line 8ϕϕ, finds the start of the record. The record number, search%, is used in the formula in line 83ϕ to calculate a PTR number. This might, of course, be an impossible number. If there are only 1ϕϕ records on a file, and you ask to see number 12ϕ, then the pointer number that is calculated by the formula will not correspond to any part of this file. Line 84ϕ checks this by using EXT#N%. This means 'extent', and it prints the byte number for the last byte on the file. The operating system can calculate this number from the catalogue information of length of file. If you have asked for an impossible number, you are politely asked again. If the number is feasible, however, line 85∅ places the pointer there, and reads in a record. To be more exact, it reads in a sector which contains the record. The data is then printed by the procedure in line 87∅, and PROCspacebar then tests for a key being pressed. If the key is the spacebar, the reading continues. If another key is pressed, you return to the menu by way of line 25∅. Note that if you were using this program in a serious way, you would want to make the ESCAPE and the BREAK keys both return to the menu without deleting the program.

Now take a look at the method of adding to the file, option 1. This calls PROCadd, in line 3∅∅. The first action here is PROCend, which finds the end of the file (line 55∅). This is done by setting the value of PTR#n% to EXT#n%, and a variable point % is also set to this value. With the pointer set to the end of the file, the data is obtained by PROCgetdata (line 56∅), and packed into one string. This is then recorded by PROCrecord (line 113∅), and then PROCupdate sets the pointer to the position that will be needed if another record is to be added. This continues until it is aborted by pressing a key other than the spacebar, and the file is closed and then opened again. The reason for this curious-looking action is so as to set EXT correctly. If the file is left open, it would be possible to read data beyond the limits of the file, using option 3, until stopped by an error. The CLOSE statement closes the file at its new length, writing an EOF byte. The file can then be reopened for further use. This only has to be done when adding to the file.

The change-data option, option 2, leads to PROCchange in line 39∅. This uses PROCfind to get the correct record, prints this with PROCprintdata, and then calls PROCchangedata in line 98∅. This allows you to change any of the items in a record, and something to note here is that the number change% is set to ∅ after the change has been made. This is because the program returns to the menu at line 23∅, and if you had a value of change% equal to 3 or 4 at this point, it would cause lines 23∅ or 24∅ to run! This is a general point about the use of BASIC, but one worth making. After the field is changed, the data is packed again into one string, and recorded at the same PTR position. There is no need to update to the next pointer position, because PROCfind is used each time the procedure repeats.

That's it! It shows what can be done with relatively simple filing methods, and provides you with a number of useful PROCs for your own use. Looked at in one chunk, it's rather daunting, but when broken down it's relatively easy to follow. I have, if anything, gone overboard on the use of PROCs, and some are used only once. By doing this, though, I have made the program easy to expand, and easier to follow.

The ultimate randomness

BBC DFS systems also permit another unusual and useful way of providing random access. This uses PTR and EXT along with two further commands, BGET and BPUT. BGET means 'byte-get', and it does just that – it gets a byte from the disk at the current pointer position. As you probably know by now, this means that a complete sector will be read, and the pointer will copy the byte from the buffer memory. The BPUT command works the opposite way round, putting a byte into the buffer and so to the disk, at the current pointer position.

Now, the advantage of BPUT and BGET is that they obey only the rules that you devise for them. You can record a string without an identifier or length byte, and the correct way round. Figure 7.11

```
100 nr%=OPENOUT"string"
110 Test$="THIS IS A TEST"
120 FOR J=0TO13
140   PTR#nr%=J
150   BPUT#nr%,ASC(MID$(Test$,J+1,1))
160   NEXT:CLOSE#nr%
```

Fig. 7.11. Using BPUT to place each byte at a position allocated by PTR.

shows how this is done, reading characters from a string, and storing their ASCII codes, one at a time, with BPUT#. Figure 7.12 shows how easily a file of this type can be read to the screen. Since VDU followed by an ASCII code will put a character on the screen, text can be placed very rapidly.

The most useful application of BPUT# and BGET#, however, is in byte files and in indexed sequential files. At this stage, we're rather

```
100 nr%=OPENUP"string"
110 REPEAT:VDU BGET#nr%
120   UNTIL EOF#nr%:CLOSE#nr%
```

Fig. 7.12. Reading a BPUT file by using BGET.

exceeding our brief in this book, but a glimpse is useful. We've seen how a set of integers can be stored in a file, with each number taking 5 bytes of storage. If you have to store numbers whose values do not stray beyond the limits of ∅ and 255, then a 'byte file' is more economical. This uses just one byte for each number, and Fig. 7.13 shows how this can be done for a set of numbers which, for the sake of illustration, is read from a DATA line. The method is to convert

```
    100 nr%=OPENOUT"bytes"
    110 FOR J%=0TO19:READ n%
    130    PTR#nr%=J%
    140    BPUT#nr%,n%
    150    NEXT
    160 CLOSE#nr%
    200 DATA2,4,6,8,10,12,14,16,18,20,1,3,
  5,7,9,11,13,15,17,19
```

Fig. 7.13. Storing integers in single-byte form. This is a very space-saving way of storing small integers (0 to 255).

the integer numbers from five-byte form into single byte form simply by the use of BPUT. Figure 7.14 shows the recovery method, with each byte printed on the screen by the BGET# statement.

```
100 nr%=OPENUP"bytes"
110 FORJ%=0TO19
120    PTR#nr%=J%
130    PRINT BGET#nr%;" ";
140    NEXT:CLOSE#nr%
```

Fig. 7.14. Reading back the file of integers, using BGET.

Finally, let's deal with indexed sequential files. Suppose you wanted to place 5000 phrases of text in memory, with random access. You could use the methods that we've discussed earlier, but if the strings vary between one character and a couple of hundred, it's wasteful to pad them all out to the maximum length. Another method is to find the length of each string, using LEN. This number will be between ∅ (null string) and 255. Two files are opened, one for characters and the other for bytes. The character file is filled, using BPUT#, for as many letters as there are in a phrase. The length of the phrase is then converted to single byte form and recorded on the other file. This is repeated until all of the file is filled. We then have a string of ASCII codes in one file, and a string of bytes in the other. How do we read them back? The byte file is read. If you want the

60th item, then you have to convert the bytes back to normal integers, and add these integers up to the 59th. That gives you the PTR# position to use in the characters file, and the 60th byte gives you the length. Set PTR# to the value of the starting number, and use the 60th byte to give the number of characters, and you can use a FOR ... NEXT loop with BGET# to read the characters. Difficult? Figure 7.15 illustrates the reading and writing in one short program just to get you started.

```
   10 REM
  100 nr%=OPENOUT"index":ch%=OPENOUT"cha
rs":K%=0
  110 FOR N%=1TO10:PTR#nr%=N%-1
  120    READ data$:L%=LEN(data$)
  140    BPUT#nr%,L%
  150    FOR J%=0TOL%-1
  160       PTR#ch%=K%:K%=K%+1
  170       BPUT#ch%,ASC(MID$(data$,J%+1,1
))
  180       NEXT:NEXT
  190 CLOSE#0
  200 STOP
  300 DATA A LOT MORE,VERY,SOME,PECULIAR
ITY,RANDOM,IS,BUT,NOT,ANTIDISESTABLISHME
NTARIANISM,PRACTICAL
  500 nr%=OPENUP"index":ch%=OPENUP"chars
":K%=0
  510 PRINT"Enter a number 1 TO 10"
  520 INPUT Z%:IF Z%<1 OR Z%>10 THEN PRI
NT"Incorrect- please try again":GOTO510
  530 PTR#nr%=0:FORJ%=0TOZ%-2
  540    K%=K%+BGET#nr%:NEXT:L%=BGET#nr%
  550 PTR#ch%=K%
  560 FORZ%=0TOL%-1
  570    VDU BGET#ch%:NEXT
  580 CLOSE#0:END
```

Fig. 7.15. Truly random files, using fields of any length up to 255 characters. Two files are used, one for the length of field, the other for the characters themselves in ASCII code form.

That, then, brings me to the end of the main part of this book, with only some tidying operations to follow. I hope that you have found much of interest, and that this spurs you on to greater appreciation of what can be achieved with any of the BBC disk systems.

Chapter Eight
DFS Commands

This is not so much a chapter, more a list of commands. The commands of all three systems are listed here in alphabetical order. The commands which are common to all three systems are unmarked, but commands which are peculiar to one system only are distinguished by [Ac] for Acorn, [Am] for AMCOM, or [Wf] for Watford. If a command is recognised by two of these systems and not by the third, this will be indicated as, for example, [NAc], meaning 'not Acorn'.

Where commands have to be followed by other information, this is also indicated by abbreviations. These abbreviations are the same as those universally used by the manufacturers, so that:

<**drv**> means drive number (Ø to 3). When this is omitted the currently selected drive will be used.

<**fsp**> means file specification, which can be a filename, a filename plus directory letter, or a filename, directory letter and drive number.

<**afsp**> means ambiguous file specification, one which can contain the * or # wild-card characters.

<**dir**> means directory letter, or other character.

The descriptions which accompany the commands are very brief, and are intended to be a reminder only of what the command does. For a full explanation, you should consult the text of this book, where relevant, or the manual for the particular DFS which you are using. A longer description is included if the command has not been dealt with in an earlier chapter of this book. Note, to avoid repetition, that no command which involves placing data on a disk will operate in a disk which has been write-protected.

The commands

***ACCESS <afsp> (L)** is used to lock or unlock a file, depending on whether the L follows the <afsp>. Early sample of AMCOM DFS did not use this command, and the AMCOM syntax is slightly different in Extended Mode.

***BACKUP <source drv> <destination drv>** is used to copy the entire contents of a disk on to another disk, regardless of what files are present. It must be enabled by *ENABLE, because memory is overwritten, and the destination disk will be completely overwritten also.

***BUILD <fsp>** allows a set of commands or text words to be entered directly. A new line number is shown on the screen after each use of RETURN, but these numbers are for reference only – they are not recorded and are *not* BASIC line numbers. Pressing the ESCAPE key terminates the *BUILD action.

***CAT <drv>** reads the catalogue for the specified drive, and displays it on the screen. If you type *CAT, then CTRL B, with a printer attached, pressing RETURN will give a printout of the catalogue. This is a very useful way of keeping track of the contents of your disks.

***CLEAR <drv> [AM]** erases a complete disk. It must be enabled by *ENABLE. No choice is allowed, except that locked files are not affected. By locking some files, *CLEAR can be used in the way that the other systems use *DESTROY.

***COMPACT <drv>** deletes unused space in the disk by moving files to occupy the lowest numbered sectors. The filenames are displayed in order as they are moved, and information on the files is displayed. AMCOM and Watford systems offer variations, but both show how many sectors are free after compacting.

***COPY <source drv><destination drv><afsp>** will copy a file or files from one drive to another. If both drives are Drive Ø, then screen messages indicate when to insert the two different disks. Using ambiguous filenames can cause several files to be copied.

***DELETE <fsp>** removes one named file from the catalogue, leaving a space that can be used by a file of the same size, or smaller. The action does not take place if the file is locked or if the disk is write-protected.

***DESTROY <afsp> [NAm]** will delete files that are specified by an ambiguous name. Pressing Y or N will determine whether or not a group of files is deleted.

***DIR<dir>** sets the directory character which will be used subsequently. This does not affect files that are already recorded. The characters #,*,., and : must not be used. The default directory character is $. The Watford DFS allows the drive number to be specified also.

***DRIVE<drv>** changes to another drive, if you have more than one drive. Drive ∅ is always selected when the machine is switched on, or when BREAK is pressed.

***DUMP <fsp>** produces a hex listing of the bytes of a file on the screen (or printer). The AMCOM and Watford systems also display the ASCII characters for code numbers that are within the ASCII range.

***EDIT** [Wf] enters the Watford disk editor program.

***ENABLE** is used to make you think twice about using any command which will overwrite a disk. If the command is not enabled, it will not be carried out, and the message 'Not enabled' will be displayed. *ENABLE affects only the command that immediately follows it.

***EXEC <fsp>** will read data from a file as if it were being entered from the keyboard. If the text is a command, the command will be executed. It is used extensively to merge programs, and in association with the !BOOT file.

***FORM <number><dry>** [Wf] is used in the Watford DFS to format a disk. The number that follows *FORM specifies the number of tracks (for example, *FORM4∅). The choice of normal or extended catalogue is made at this time.

***FORMAT <drv> [Am]** is used in the AMCOM system to format a disk. The setting of 40- or 80-track is done by the *OPT2 and *OPT3 commands. It must be enabled by *ENABLE, because it completely removes any previous data on the disk.

***FX11∅,<drv>,n** [Wf] sets or clears double-step mode. Setting double-step mode for an 80-track drive allows 40-track disks to be read or written. The number n should be 255 to set, and ∅ to clear this mode.

***FX111** [Wf] is used in conjunction with assembly language to find the drive number of the last *LOAD, *RUN, LOAD or CHAIN.

***FX199** is used in conjunction with assembly language to find the channel number of a file that is being SPOOLED.

***HELP <word>** is used in all DFS systems to remind the user of the syntax of commands. The AMCOM system keeps this information on a utility disk. The Watford system has two other *HELP commands, listed separately below.

***HELP FILES** [Wf] lists all files that are currently open, along with channel numbers and PTR values.

***HELP SPACE <drv>** [Wf] lists the spaces that have been left on a disk by deleting files. This indicates how much extra space would be made available by using *COMPACT. The total free space is also shown.

***INFO <afsp>** gives information on a file or a set of files. The information is, reading from left to right, the directory letter, name, lock status, load address, execution address, length in bytes and start sector. These numbers are in hex. The AMCOM DFS gives the drive number also.

***LIB:<drv><dir>** sets the library character on a given drive. Any machine code files that are subsequently *SAVEd will use this library character, and can be loaded and run by typing an asterisk immediately followed by the filename, such as *INDEX.

***LIST <fsp>** prints out a text file in the form of characters, allocating numbers to each line as it is displayed. Characters which are non-ASCII can cause odd effects. The AMCOM DFS will print a dot for each non-ASCII character, so that such characters do *not* cause trouble.

***LOAD <fsp><address>** will load a machine code program to a stated address. The address must be in hex, with no '&' mark. Any other group of bytes can also be loaded in this way, provided an appropriate address is used. *LOAD <fsp> 8ØØØ will have the effect of 'loading' a file to the ROM (which is unaffected), so that this can be used to check that a file is not damaged. The Watford DFS can make use of an ambiguous filename, in which case the first file which answers to the description will be loaded.

***MLOAD <afsp>** [**Wf**] will load a file into the disk RAM area, to address &12ØØ. The DFS is then switched off, and the program is loaded to its correct address, which may be lower (typically &ØEØØ). This allows programs to be used from tape. The commands *HELP SPACE and *HELP FILES must *not* be used following this command, as they would corrupt this area of memory.

***MOVE <source drv><destination drv><afsp>** [**Wf**]is a form of *COPY which allows a number of files to be copied. For each file, a Y or N can be entered to determine whether that file will be copied or not. It will overwrite memory, as the *COPY command does.

***MRUN <afsp>** [**Wf**] produces the effect of *MLOAD followed by *RUN. The file is loaded to address &12ØØ, and the DFS is then switched off. The file is then moved to its correct load address, and executed. The execution address that has been saved on the file will be used.

***OPT1,n** enables or disables the messages that are displayed when a file is loaded. Using Ø disables messages, any other number from 1 to 99 enables. A space can be used to replace the comma between the '1' and the following number.

***OPT2,n [Am]** selects the number of sectors per track. This will allow the AMCOM system to make use of 8-inch disk drives, which use 18 sectors per track. The normal ten sectors per track is reselected on pressing BREAK, and when the machine is switched on.

***OPT3,n [Am]** selects the number of tracks. This will usually be 40 or 80, but 8-inch drives have 72 tracks, and a few disks used mainly on other machines have 35 tracks. This, and *OPT2, permit a large range of drives to be used. The importance of this is that 8-inch disks make use of the universal IBM standards, so that such disks are freely interchangeable among machines. After pressing BREAK, or on switching on, the number of tracks will be set to 40 or 80, depending on the setting of the links on the keyboard.

***OPT4,n** controls the auto-booting options. When SHIFT-BREAK is pressed (see Chapter 4), the file !BOOT will be checked in Drive Ø. The number, n, in *OPT4,n then determines what is done with this file if it exists. If n is Ø, the file is ignored. If n is 1, the file is *LOADed. If n is 2, the file is *RUN, and if n is 3, the file is *EXECuted.

***OPT5,n [Am]** sets an address in memory. Any addresses lower than this will *not* be used by *BACKUP,*COPY or *COMPACT. In this way, a program can be protected from the effects of these commands. The number n is the number of 256-byte blocks of address number.

***OPT6,n [Am]** allows a wide choice over what information is displayed when a file is loaded. It must be enabled by *OPT1,1.

***OPT7,n [Am]** is used in conjunction with *OPT5,n. When the address in memory has been set by *OPT5,n, the use of *OPT7,n then specifies how many sector-sized blocks (256 bytes) of memory can be used by the commands *BACKUP, *COPY and *COMPACT.

***OPT8,n [Am]** will allow a 40-track disk to be used on an 80-track drive by using n=255. Using n=∅ reverts to normal operation.

***RENAME <old fsp><new fsp>** is used to change the filename of a file that is already on a disk. The directory letter can be changed as well, but the drive cannot be changed. The new fsp must not be one that already exists. If it does, a 'File exists' message will appear. The Watford system allows ambiguous filenames, so that a group of files can be renamed.

***RUN <fsp> other descriptions** will load and run a machine code program. Data parameters, or other program names can be added to the command to force the machine code program to make use of these additional items.

***SAVE <fsp>start finish execute reload** is used to save a block of bytes or a machine code program. The execute and reload addresses (in hex) are optional but, if a reload address is used, an execute address must also be provided. This can be identical to the load address.

***SPOOL <fsp>** will save a file on the disk of all the information that is produced on the screen by LIST or PRINT commands. The disk file will consist purely of ASCII characters, with no BASIC 'tokens' or control numbers. The use of *SPOOL creates a file that can be read by a text editor/word processor program, or by *EXEC. The action is ended by typing *SPOOL with *no* filename. *SPOOL and *EXEC can be used to merge BASIC programs.

***SYS n [Am]** is used in the AMCOM system to select the Acorn-compatible operation (n=∅) or the extended AMCOM mode (n=1). An Acorn disk with a coloured title will not load until a utility, provided on the AMCOM utility disk, is used.

***TITLE name** will title a disk, giving it (or changing) a reference name of up to twelve characters. If the title includes spaces, it must be surrounded by quotes.

***TYPE <fsp>** displays a text file on the screen. This is the same action as *LIST, but no line numbers are displayed. The AMCOM system replaces non-ASCII characters by a dot.

***VERIFY <drv> [Wf]** examines each sector of a disk to check that the contents appear to be correct. There is a 'checksum' number recorded on each sector which can be used to determine whether or not the data is corrupted.

***WIPE <afsp>** allows a file or group of files to be deleted. For each filename that is presented on the screen, you are given the option of typing Y to delete or N to retain. Locked files are not deleted.

***WORK <:drv><dir>.name [Wf]** sets a filename, or complete specification, that will be used if no other filename is given. This allows commands such as SAVE"" to be used, and *RENAME can also be used to rename a file to the *WORK name. The 'workname' is shown separately in the catalogue of the Watford DFS.

Index

STREAMLINE YOUR COMPUTING!

The costs of disks are now comparable to cassettes, and the prices of disk drives – once a luxury for the few – are falling fast.

Consider all the advantages of disk systems, including:

- **High speed of operation.**
- **Random access to any point on the disk.**
- **One program can call another, enabling you to deal with much more complex operations.**
- **The ability to run long programs, up to many times the memory capacity of your micro.**
- **Parts of the disk itself can be edited or altered to retrieve damaged programs and read disks made on other machines.**

This book sets out all the advantages of using disks, explains the jargon and covers the principles of disk operations, not only for the BBC disk system but also the AMCOM, the Watford and other systems suitable for use with the BBC Micro. Normal and advanced disk operations are described in detail and you are shown how to take full advantage of the various disk utilities. You will be amazed at how efficient and creative your computing will become.

The Author
Ian Sinclair is a well-known contributor to journals such as *Personal Computing World, Computing Today, Electronics and Computing Monthly, Hobby Electronics,* and *Electronics Today International.* He has written over forty books on electronics and computing aimed mainly at the beginner.

GRANADA PUBLISHING
Printed in Great Britain 0 246 12325 7

£6.95 net

SINCLAIR DISK SYSTEMS FOR THE BBC MICRO GRANADA